

INTELLIVISION™

BLACK WHALE



Reverse Engineering

Report

23-Apr-2018, B

Copyright © 2018 — Joseph Zbiciak, Frank Palazzolo
Intellivision™ is a trademark of Intellivision Productions, Inc.

Intellectual Property Notice

This document represents the efforts of Joseph Zbiciak and Frank Palazzolo to reverse engineer and document this unique piece of Intellivision development history.

At the time they were created, the artifacts described were the property (both intellectual and physical property) of Mattel Electronics.

Today, *Intellivision Productions, Inc.* retains the intellectual property rights to the hardware design and program code described within this document. *Intellivision Productions* also holds the *Intellivision*[™] trademark.

Neither Joseph Zbiciak nor Frank Palazzolo is affiliated with with *Intellivision Productions, Inc.*

Background	6
The Electronics	7
Circuit Board 1: The Black Whale Monitor Board	7
Component side	7
Solder Side	8
Schematic	9
Parts List	10
Observations	10
Circuit Board 2: The Black Whale Serial Port Board	11
Component side	11
Solder Side	12
Schematic	13
Parts List	14
Observations	14
Other Circuits	15
Cartridge RAM	15
The Keyboard Component	18
Expansion Port Pinout	18
The Master Component	20
The Debug Monitor User's Guide	21
Preliminaries: Setting up the Environment	21
Command Format	21
Monitor Command List	22
CP1610 Download Record Format	23
ASCII to Nickel Conversion Chart	23
Download Record Fields	23
Download Record Encoding Example	24
6502 Download Record Format	26
The Reverse Engineered Black Whale Monitor Code	27
High Level Hardware Memory Map Summary	27
ROM Cartridge	27
Serial Cartridge	27
Keyboard Component	27
6502 Side	27
CP1610 Side	28
Black Whale Monitor, 6502 Side	29
Important 6502-side variables	29

6502 ➡ CP1610 command list	29
Subroutine / branch target list	30
\$E000: Launching the Monitor	33
\$E04C: Delay ~1 second and stuff “X\r\n” into keyboard input FIFO	34
\$E069: Menu Item for ‘X’ copied to CP1610 side	35
\$E08E: Main Initialization	35
\$E0BD: Top of monitor command loop: Send out CR+LF and ‘>’ prompt.	36
\$E0EB: Zero out rightmost argument and re-enter command loop	36
\$E0F3: Receive command byte, decode and dispatch	37
\$E150: Bad Command: Send BEL, ‘>’, CR+LF and start over	38
\$E163: Second-level dispatches for monitor commands	38
\$E195: Receive the “ZERO” command	38
\$E1B4: Process the ‘Y’ command: Resume CP1610 execution at given address	39
\$E1D0: Dispatch to ‘G’ vs. ‘GO’ command	39
\$E1E1: Process ‘GO’ command	39
\$E1E4: Process ‘G’ command	39
\$E216: Run code on CP1610 side, and poll for requests from it, until CP1610 halts	40
\$E25E: Process R / RR Command	41
\$E2A2: Show details of where we halted	42
\$E310: Process ‘P’ command: Proceed with execution at current PC	43
\$E31F: Process ‘T’ command: Run ‘T’o address, or single s’T’ep	43
\$E32E: Process ‘M’ command: Print 8 memory values from CP1610 side	43
\$E382: Process ‘N’ command: Like ‘M’, but steps backward through memory	44
\$E3AA: Dead code	45
\$E3B0: Process ‘/’ and ‘;’ commands: Inspect and replace values in memory	45
\$E470: Process the ‘S’ command: Search for 8 or 16-bit values in CP1610 memory	47
\$E4C1: Process the ‘V’ command: Verify a region by computing checksums	48
\$E58E: Process the ‘ZERO’ command: Zero out \$5000 - \$6FFF, \$9000 - \$9FFF	51
\$E5E4: Validate hex digit, and push into arg at \$06 - \$07; return C=1 on error	51
\$E615: String: “\024STARTING DOWNLOAD...”	52
\$E62A: Process a download record (‘:’)	52
\$E666: Process the checksum on a download record (‘;’ portion)	53
\$E6A8: Receive next declc, encoded as 2 chars, into \$0E:\$0D	53
\$E6C6: Get record address into \$0C:\$0B, or switch to Intel HEX processing	54
\$E6FB: Get next nickel of record being downloaded to us; handle ‘:’ and ‘;’ also	55
\$E71C: Handle ‘:’ in download record: Unwind stack and restart record	55
\$E723: Handle ‘;’ in download record: Unwind stack, receive & validate checksum	55
\$E72A: Unwind stack and report “BAD RECORD...”	55
\$E732: Process Intel HEX record for 6502-side	56

\$E783: Get hexadecimal byte from serial port (blocking)	56
\$E799: Get hexadecimal digit from serial port (blocking)	57
\$E7B2: Print X:Y to the CRTC at \$B820 - \$B823	57
\$E7D3: Convert upper nibble of A into a hexadecimal digit (ASCII)	58
\$E7D7: Convert lower nibble of A into a hexadecimal digit (ASCII)	58
\$E7E5: Print "RECORD OK... " to CRTC at \$B806.	58
\$E7F0: Print "BAD RECORD... " to CRTC at \$B806	58
\$E7FB: Update download error count on the screen at CRTC \$B81D - \$BD1E	58
\$E81D: Various status strings	59
\$E85C: Process 'U' command: Upload data from CP1610 to the host	59
\$E946: Send single hex digit as one of "@ABCDEFGHIJKLMNO" out serial	61
\$E94E: Convert upper nibble of A into a hexadecimal digit (ASCII)	61
\$E952: Convert lower nibble of A into a hexadecimal digit (ASCII)	62
\$E960: Launch the CP1610 side of the debugger; \$0D holds success/failure	62
\$E9A3: Copy CP1610 code from \$8000-\$8FFF (6502) to \$8800-\$8FFF (CP1610)	63
\$E9CC: Set up addresses for copying CP1610 code from EPROM to RAM	63
\$E9ED: Delay for a long time, proportional to 'A' squared.	64
Interlude: R6551 Details	64
\$EA01: Device Service Record (DSR) for R6551 serial port.	66
\$EA16: Initializer function for R6551 serial port, called during DSR installation	66
\$EA26: DSR interrupt handler for R6551 serial port	66
\$EA8C: Set up DSR for R6551 serial port	67
\$EAA4: Send a CR+LF out the serial port (blocking)	68
\$EAA9: Send an LF out the serial port (blocking)	68
\$EAAB: Send the byte in 'A' out the serial port (blocking)	68
\$EABC: Block waiting for a byte from the serial port; return byte in A and \$05	68
\$EAC9: Test whether data is waiting on serial port; C=1 means "yes"	68
\$EAD2: Receive byte from serial port, non-blocking, with XON/XOFF flow control	69
\$EAF1: Copy string at X:Y, length A, to CRTC frame buffer at \$B806	69
\$EB06: Print 10-bit hex value via serial port (blocking)	69
\$EB0E: Print 8-bit hex value via serial port (blocking)	69
\$EB17: Convert nibble to ASCII hex and send to host (blocking)	70
\$EB30: Process commands that start with '\$': \$M, \$B, \$C, \$R, \$D	70
\$EB4F: Process '\$B' command: Set breakpoint at address	70
\$EB61: Process '\$C' command: Clear all breakpoints	70
\$EB69: Process '\$R' command: Remove breakpoint at address	71
\$EB7B: Process '\$D' command: Display breakpoints	71
\$EBA9: Process '\$M' command: Memory copy	71
\$EC00: Send CMD in 'A' to CP1610 side; spin until CP1610 indicates it's done	72

\$EC09 - \$EFFF: Empty ROM	72
\$F000 - \$FFFF: Intellivision BASIC ROM	72
Black Whale Monitor, CP1610 Side	73
Important CP1610-side variables	73
Subroutine / branch target list	74
\$8800: Initialize CP1610 side of debugger	76
\$887B: Main command dispatch loop	77
\$88C5: CMD #\$01: Copy R3 words from argument buffer @R4 to @R1	78
\$88CF: CMD #\$02: Display string from argument buffer @R4; Length in R3	78
\$88E9: CMD #\$03: Read a decl @R1, and write to argument buffer @R4	78
\$88ED: CMD #\$04: Read a 16-bit value @R1 and write to argument buffer @R4	78
\$88F4: CMD #\$05: Write a 16-bit value from argument buffer @R4 to loc @R1	79
\$88FA: CMD #\$06: Resume execution at address in R1	79
\$8944: CMD #\$07: Disable all breakpoints	80
\$8951: CMD #\$08: Clear breakpoint at address in R1	80
\$8970: CMD #\$09: Set breakpoint at address in R1	81
\$89A0: CMD #\$0A: Copy breakpoint address list to 6502 side	82
\$89BD: CMD #\$0C: Copy saved CP1610 registers to 6502 side	82
\$89DA: CMD #\$0D: Set saved register value; Reg # in R3, value in R1	83
\$89F4: CMD #\$0E: Single step instruction, or run to address in R1 if R1 ≠ 0	83
\$8A26: CMD #\$0F: Resume execution at currently saved PC	84
\$8A2E: Get arguments for command	84
\$8A3F: Clear the command byte and return to the command dispatch loop	85
\$8A47: Alias for \$8A3F	85
\$8A4A: Copy original code to breakpoint buffer and install breakpoints	85
\$8A65: Set up SIN2 on the other side of the next instruction (single step)	85
\$8C04: Restore machine state and prepare to jump into the program	93
\$8C2B: Interrupt service routine: Detect breakpoints	93
\$8CC0: Re-enter command dispatch loop; override any command request w/NOP	95
\$8CD1: Resume execution after single step	96
\$8CF1: Capture some state to prepare to single-step over a breakpoint	96
\$8D04: Branch to default EXEC ISR at \$1126	97
\$8D07: Replace breakpoints with original opcodes/data	97
\$8D1F: Restore opcodes after a single step	97
What we still don't know	98
Some tantalizing hearsay	98

Background

The Blue Sky Rangers posted a history of the Keyboard Component, aka. Blue Whale,¹ over at [the Intellivision Lives! website](#). While the Keyboard Component failed as a product, its story has an interesting denouement, as described at the link above:



FUN FACT: All the money spent on the Keyboard Component didn't go **totally** to waste; as the ranks of the programmers swelled during 1982, development systems could not be assembled fast enough. The bottleneck was the hand-built **Magus board**—the interface between the development computer and the Intellivision. Then some bright person realized that a Blue Whale would make a dandy development system! A slightly modified Keyboard Component (dubbed a **Black Whale**) would accept the compiled object code for a game serially to its internal RAM; that RAM, mapped to the same addresses as an Intellivision cartridge, would be read by the attached Master Component as if it **were** a cartridge.

While slower to download a game than a Magus board (which read data from the computer over parallel lines), the Black Whale proved to be a quick, cheap solution. By the middle of 1983, half of the development systems in Hawthorne and all of the development systems at [Mattel Electronics France](#) used Black Whales.

For years, little more was known publicly about the modifications that defined the Black Whale. Various hints and details dribbled out through the years, but very little concrete information. *Until...*

Daniel Bass [revealed in February 2018](#) that he has the two key circuit boards that turn a Blue Whale into a Black Whale. Joseph Zbiciak arranged for short-term access to these boards in order to capture their details for posterity, and to fill this gap in our knowledge of Intellivision development.

The reverse engineering team primarily consists of Joseph Zbiciak and Frank Palazzolo. This document was written primarily by Joseph Zbiciak. The remainder of this document includes:

- A detailed description of the actual circuit boards that form the Black Whale modification.
- The necessary modifications to the Intellivision system to fully utilize the Black Whale.
- How to use the Black Whale.
- A detailed disassembly of the monitor software embodied in the Black Whale.
- What we still don't know.

Follow along as we dig into this interesting piece of Intellivision history.

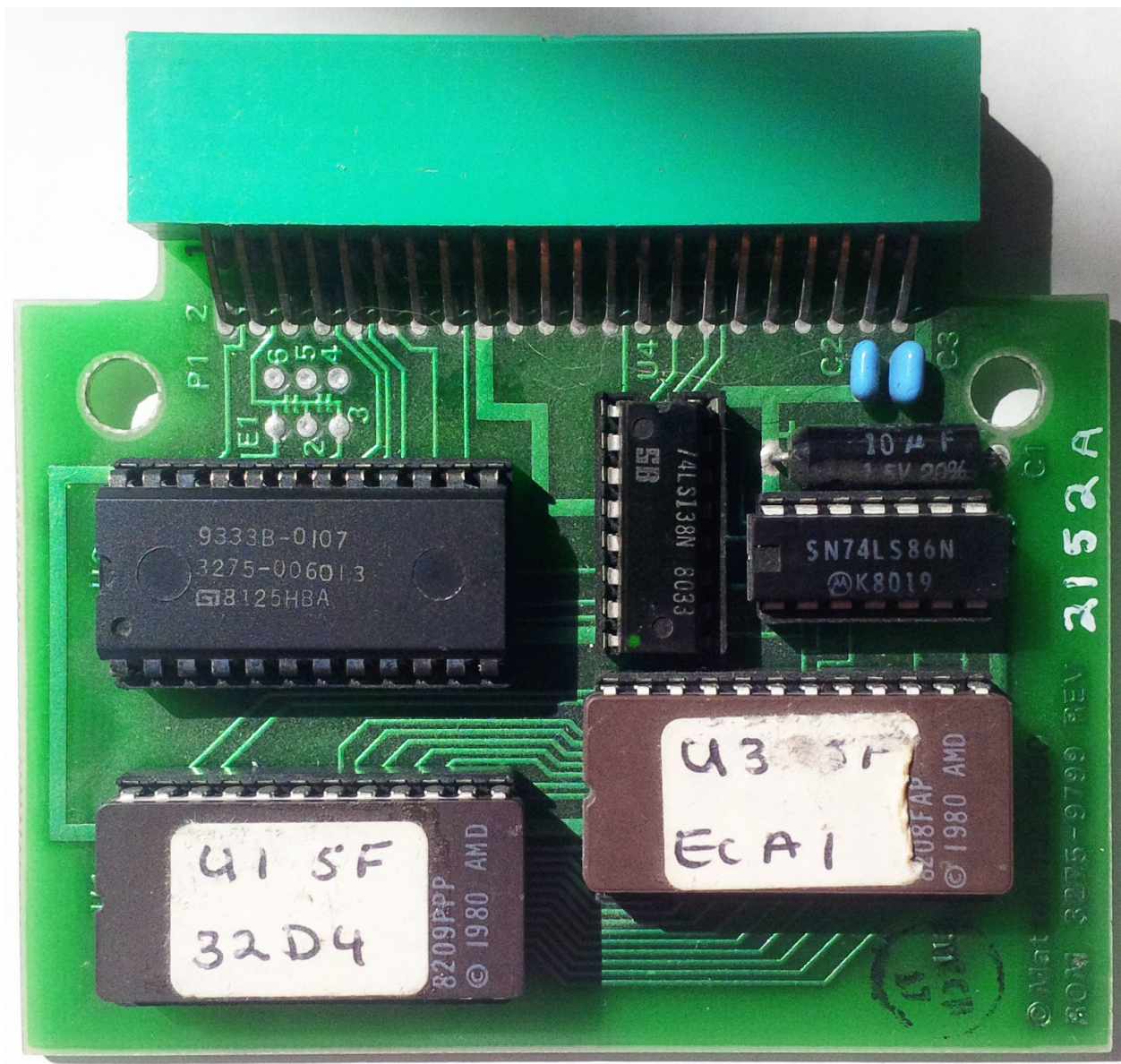
¹ Steve Roney informs me that the Blue Whale/Black Whale naming may be inaccurate. So far, nobody who was present at the time that's been asked remembers Black Whale, in particular. Perhaps the story quoted above is apocryphal. Take the name itself with a grain of salt, and read up on the tech.

The Electronics

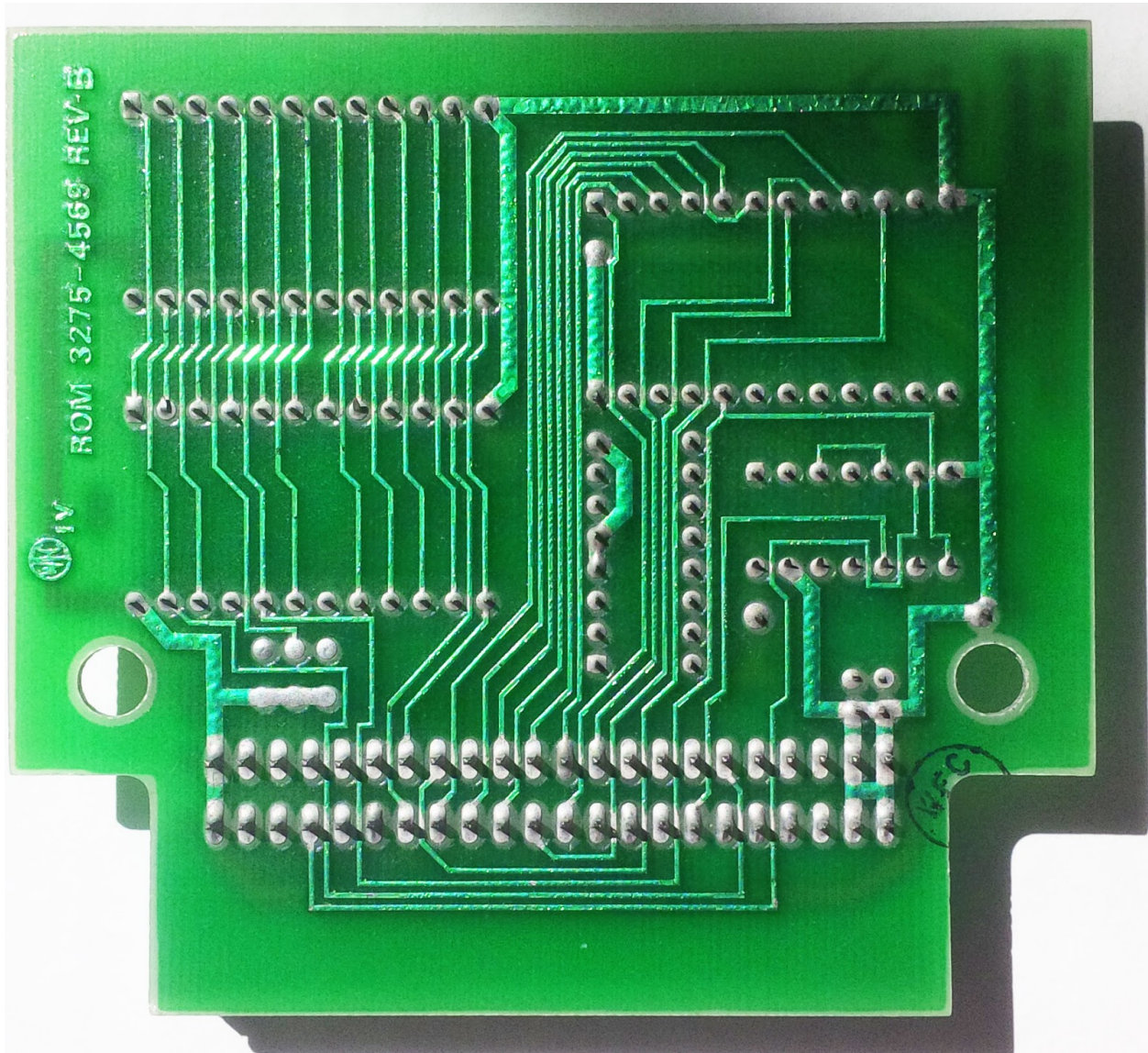
Circuit Board 1: The Black Whale Monitor Board

This circuit board is actually a Keyboard Component BASIC cartridge board that still has one of the Keyboard Component BASIC ROMs intact. The other two ROM positions are filled with EPROMs containing the Black Whale Monitor program.

Component side

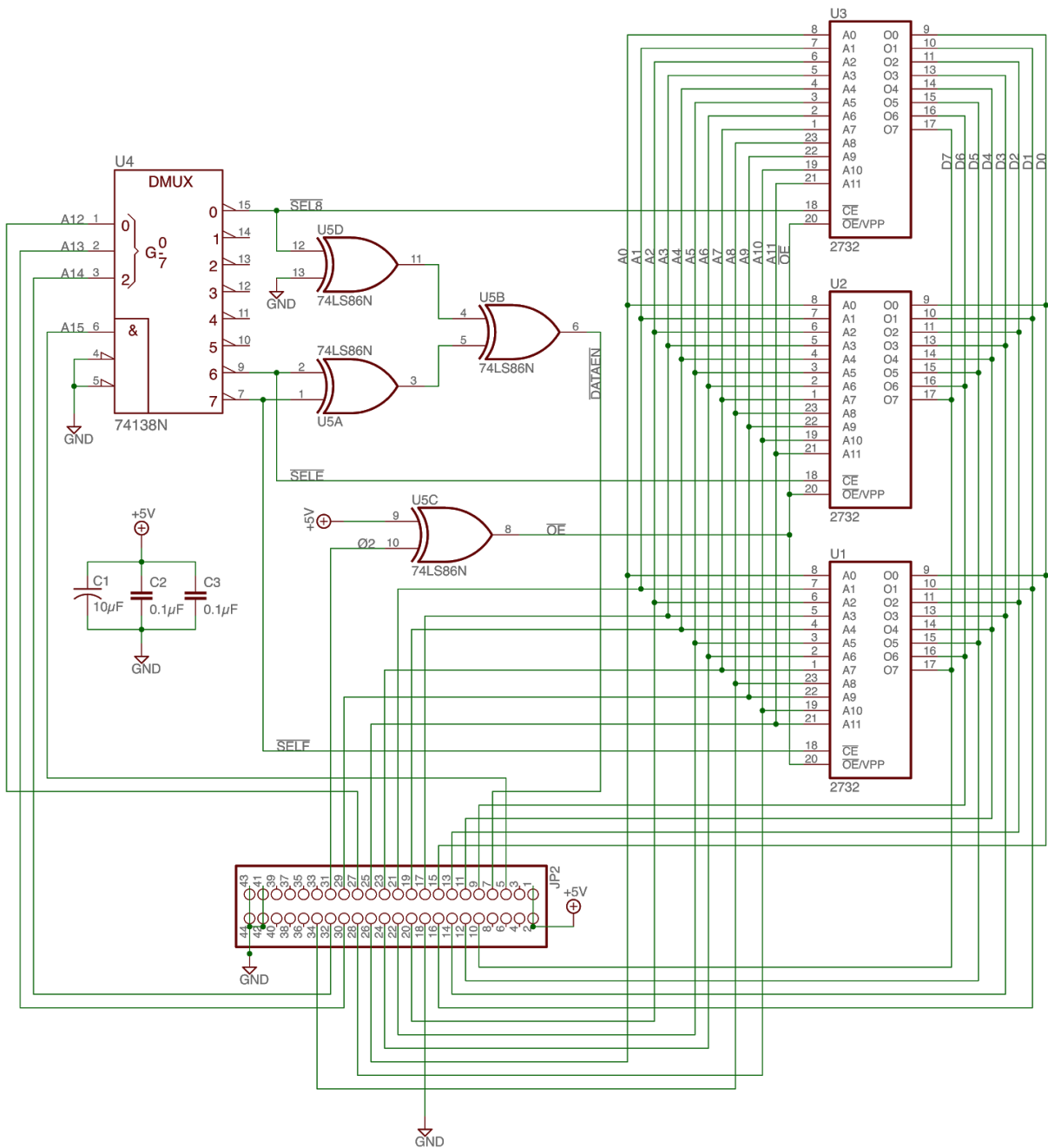


Solder Side



This is a straightforward ROM cartridge board that's been repurposed to deliver the Black Whale monitor to the Keyboard Component and Master Component.

Schematic



The monitor board schematic is fairly straightforward. As indicated in the parts list, the three ROMs U1, U2, and U3 map to $0xE000 - 0xEFFF$, $0xF000 - 0xFFFF$, and $0x8000 - 0x8FFF$, respectively.

See [The Keyboard Component :: Expansion Port Pinout](#) below for the connector pinout.

Parts List

- U1: 2732 EPROM (U1 5F 32D4), mapped at 0xE000 - 0xEFFF.
- U2: Keyboard BASIC upper ROM, PN 9333B-0107, mapped at 0xF000 - 0xFFFF.
- U3: 2732 EPROM (U3 ?F ECA1), mapped at 0x8000 - 0x8FFF.
- U4: 74LS138 3-to-8 decoder.
- U5: 74LS86 Quad XOR.
- C1: 10 μ F electrolytic capacitor, axial.
- C2, C3: 0.1 μ F capacitors.

Observations

The top side of the board gives the part number as ROM 3275-9799 REV 2152 A (where 2152 A is hand-written in white ink), while the bottom side gives the part number as ROM 3275-4569 REV B.

The 74LS138 (U4) performs a rather straightforward address decode of the upper four address lines (A15..A12). It produces three active-low signals (labeled /SELF, /SELE, and /SEL8) that go to the corresponding /CE inputs on U2, U1, and U3 respectively.

The monitor board has an interesting set of customization pads next to U2.

Pad	Connection
E1	A11 pin on U2
2	A11 pin on U1
3	A11 pin on U3
4	V _{CC}
5	V _{CC}
6	V _{CC}

It appears the purpose of these pads and the cut-points is to allow using 2716 ROMs in some sockets by cutting at the designated cut point and tying the corresponding pad to V_{CC}.

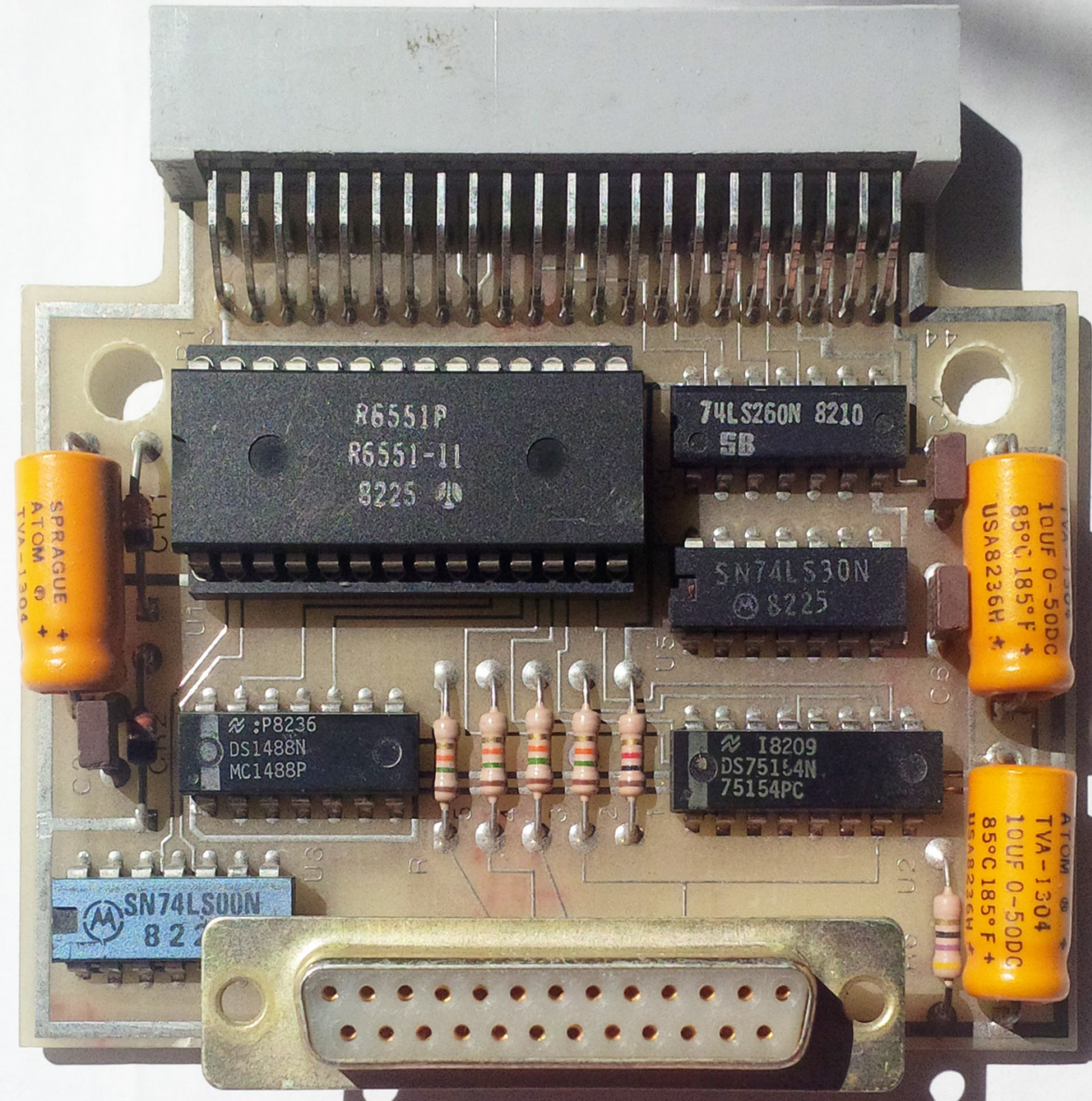
The /OE signal for the three ROMs is just the inverse of the ϕ 2 clock signal.

The ROM board does not look at the R/W signal (pin 6), and so cannot support RAM. In fact, it will engage in buffer fights if you try to issue a write to an address it "owns."

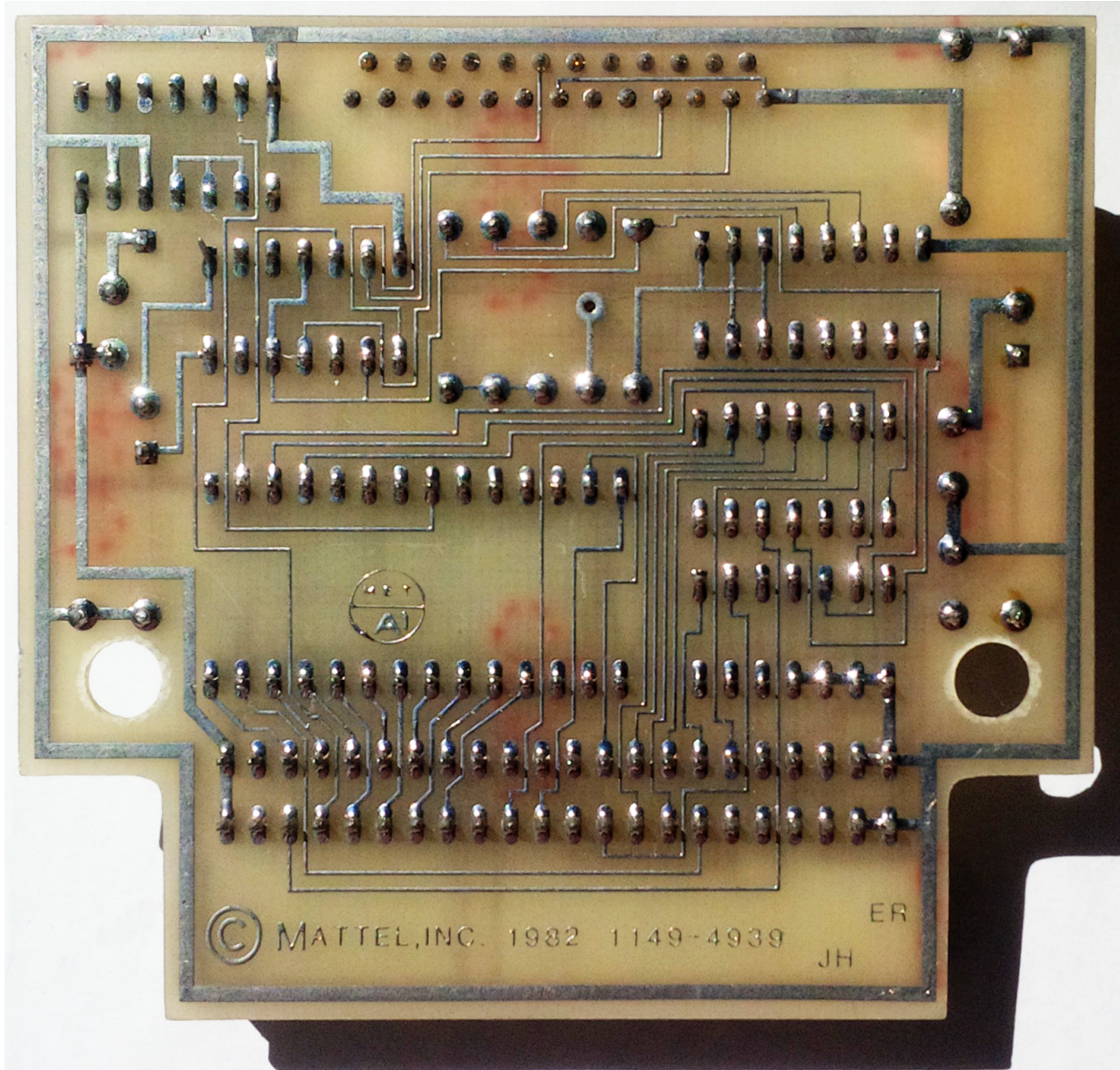
The pins on U2 have some sort of black coating on them.

Circuit Board 2: The Black Whale Serial Port Board

Component side

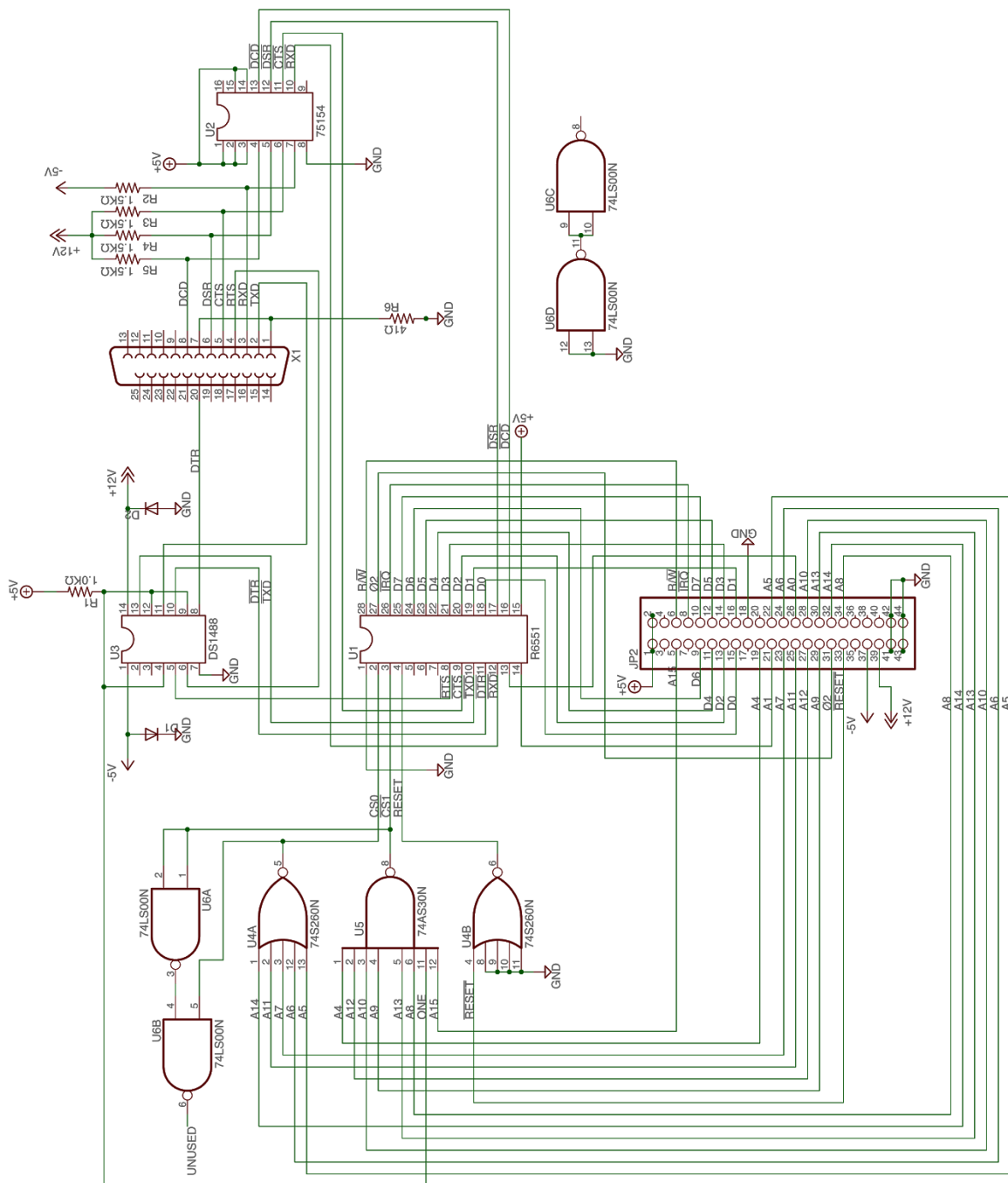


Solder Side



This is a pretty straightforward serial port interface that is driven by the monitor program contained on the first board.

Schematic



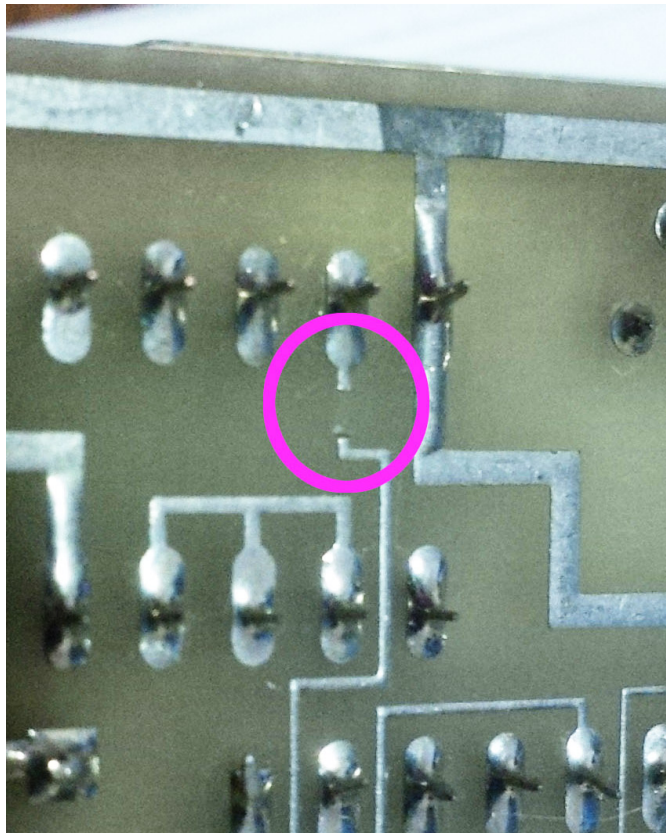
See [The Keyboard Component :: Expansion Port Pinout](#) below for the connector pinout.

Parts List

- U1: Rockwell R6551 ACIA, mapped at \$B710 - \$B713 (w/ aliases through \$B71F).
- U2: 75154 Quad Line Receiver
- U3: DS1488N Quad Line Driver
- U4: 74LS260 Dual 5-Input NOR
- U5: 74LS30 8-Input NAND
- U6: 74LS00 Quad 2-Input NAND
- C1, C3, C5: 10 μ F high temp, high voltage electrolytic capacitors
- C2, C4, C6: 0.1 μ F capacitors (ceramic?)
- R1: 1k Ω 1/2W resistor
- R2, R3, R4, R5: 1.5k Ω 1/2W resistor
- R6: 41 Ω 1/2W resistor
- CR1, CR2: Diodes. (Silicon switching diodes?)

Observations

The output from the 74LS00 originally drove the /DATAEN signal on pin 7. However, its trace was cut, which means the serial board does not use /DATAEN to claim address space, and in fact apparently must avoid it:



The serial voltages are +12V and -5V, as that's the only two voltages provided by the KC that are at all appropriate. The +12V is kosher. The -5V is a bit marginal.

The serial board is wired up to assert /IRQ. However, analysis of the monitor ROM suggests they gave up on interrupt-driven I/O.

The R6551 uses $\phi 2$ as its clock input. The expected baud rate is just shy of 9600 baud, as $\phi 2$ is 895kHz on NTSC machines. (Note: $\phi 2$ should be connected to XTLI (pin 6) of the R6551; however, I forgot to check that connection before relinquishing the boards.)

Other Circuits

Cartridge RAM

Some commentary by Keith Robinson suggests the Keyboard Component itself may have been modified to map RAM at \$5000 - \$6FFF. Alternately, a photo from [Pascal Bernard's interview of Patrick Aubry about Mattel Electronics France](#) suggests a T-Card was used to provide the necessary RAM. If you look closely at the Keyboard Component, you can see a T-Card-sized item hanging from the side:



The T-Card is capable of accepting up to 8K words of RAM by setting its DIP switches as follows², and populating all EPROM/RAM sockets with 6116 SRAMs:

DIP Position	Name	Bank 0	Bank 1	Bank 2	Bank 3
8	A15	ON	ON	ON	ON
7	A14	off	off	off	off
6	A13	ON	ON	off	off
5	A12	off	off	ON	ON
4	RAM	off	ON	off	ON
3	RAMH	ON	off	ON	off
2	ENA	off	off	off	off
1	NC	ON	ON	ON	ON

Note that the switch settings look inverted from what you might expect: “ON” means logical zero, while “off” means logical one.

² Not tested, but believed to be correct.

The Keyboard Component

As far as I can tell from examining the monitor disassembly and so forth, no modifications to the Keyboard Component are necessary to convert a Blue Whale to a Black Whale as long as a T-Card provides the necessary RAM.

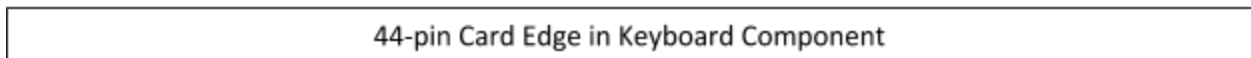
The monitor makes use of the Keyboard Component's high resolution text overlay for certain status messages during download and debug. It does not make use of the tape drive or the actual alphanumeric keyboard itself at any point during debug. Rather, the debugger is driven entirely via the serial link.

Expansion Port Pinout

Currently, we do not have a complete schematic of the Keyboard Component. Frank Palazzolo has traced out most of the expansion port signals however, producing the following pinout. Looking at the KC from the back, the pins are arranged as follows: Odd-numbered pins on the bottom, even numbered pins on the top, and numbering starts from the left.

2

44



1

43

Currently known pin assignments:

Signal	Pin # (Bottom)		Pin # (Top)	Signal
V _{cc} (+5V)	1		2	V _{cc} (+5V)
/RD_8000_BFFF	3		4	/WR_8000_BFFF
A15	5		6	R/*W
/DATAEN	7		8	/IRQ
D6	9		10	D7
D4	11		12	D5
D2	13		14	D3
D0	15		16	D1
A3	17		18	/ADDR_CTRL_EN
A4	19		20	A2
A1	21		22	A5
A7	23		24	A6
A11	25		26	A0

A12	27		28	A10
A9	29		30	A13
$\phi 2$	31		32	A14
RESET	33		34	A8
?	35		36	?
V_{BB} (-5V)	37		38	?
V_{DD} (+12V)	39		40	?
GND	41		42	GND
GND	43		44	GND

The /ADDR_CTRL_EN must be pulled low in order for the cartridge to see any of the other control lines. Both Black Whale expansion boards tie this signal to ground.

The /RD_8000_BFFF and /WR_8000_BFFF signals are not used by either board. These signals also get disabled by the KC if the cartridge asserts /DATAEN. It's unknown what purpose these signals serve.

The /DATAEN signal must be driven low to claim the address space \$E000 - \$FFFF. Otherwise, the KC EXEC ROM aliases into that address space. The ROM board asserts /DATAEN for all three ROM spaces (\$8000 - \$8FFF, \$E000 - \$EFFF, \$F000 - \$FFFF).

The V_{BB} signal (-5V) does not have a large current carrying capacity. Its trace width is the same as a digital signal trace width. *Do not draw large currents through this.* The serial board uses this voltage for the negative side of its RS232 swing. If someone makes a modern version of this board, I'd suggest using a charge-pump based circuit (e.g. [MAX232](#)) and avoid using V_{BB} . V_{BB} is likely intended for the V_{BB} inputs on the [4116 DRAMs](#) which require -5V on that input.

The $\phi 2$ clock rate is Colorburst / 4 (894.886kHz) on NTSC. This affects the baud rate generated by the serial board. The serial board uses this signal for baud rate generation. A standard BRG crystal is 1.8432MHz. The ~895kHz clock can generate baud rates within about 3% of standard baud rates.

The RESET signal (pin 33) is inverted relative to the RES input on the 6502 (pin 40). This makes the signal active high / falling edge triggered. That is, when RESET = 1, everything is held in reset, and when RESET goes 1 => 0, the processor begins its reset sequence. The serial port board inverts this signal before driving it to the M6551.

The Debug Monitor User's Guide

The Black Whale provides a rather featureful debug monitor. This section describes how to use the debug monitor, based on what we've reverse engineered.

Preliminaries: Setting up the Environment

Open up your Master Component, and connect a clip-lead between pin 28 and pin 40 of the CP1610 processor. This is necessary to enable breakpoint functionality. Now close up your Master Component and seat it in the Keyboard Component.

Insert a T-Card configured for 8K x 16 RAM at \$5000 - \$6FFF and insert it in the game cartridge slot of the Keyboard Component.

Insert the Black Whale cartridges in the two expansion slots on the back of the Keyboard Component. It does not matter which cartridge goes in which slot, but both are required.

Connect a serial terminal with the ability to upload from local disk³ to the Black Whale Serial Board. Configure the terminal for 9600-8-N-1⁴.

Connect the Master Component + Keyboard Component to a TV or suitable monitor. You are now ready to begin developing with your Black Whale.

Command Format

The Black Whale monitor commands fit a very simple, basic format:

- Up to 3 hexadecimal arguments separated by commas.
 - An argument may be omitted in-place by providing just a comma.
 - A less-than sign '<' may be used in place of a comma.
 - The number of digits in an argument is sometimes significant.
- A command verb.
 - Most command verbs are single characters.
 - Some are multiple characters.
- A newline.

No whitespace is permitted in a command. If an invalid character is received, the monitor will beep and reset the command input. All commands and values must be in UPPERCASE.

³ The images need to come from somewhere!

⁴ Based on the divisor programmed into the R6551, and the ϕ 2 clock rate of ~895kHz. See [SEA16: Initializer function for R6551 serial port, called during DSR installation](#).

In addition to issuing commands, you can also directly initiate a code download to either the CP1610 or 6502 address space. The download image formats are described later. There is no explicit download command; rather, the monitor recognizes download records and processes them as it receives them.

Monitor Command List

Command	Description
ZERO	Zero out game memory \$5000 - \$6FFF, \$9000 - \$9FFF
<i>addr</i> Y	CP1610: Resume execution at <i>addr</i> .
G	Go to Game. CP1610: Jump to \$1041. 6502: Erase text overlay.
<i>addr</i> G	Go to Game at <i>addr</i> . Like 'G' but with a different target address.
<i>addr</i> GO	Go to <i>addr</i> on 6502 side.
<i>reg</i> , <i>val</i> R	Update register <i>Rreg</i> to <i>val</i> on CP1610 side.
RR	Show current register values on CP1610 side.
P	'P'roceed with execution at current breakpoint.
<i>addr</i> T	Run 'T'o <i>addr</i> on CP1610 side.
T	Single s'T'ep over current instruction on CP1610 side.
<i>addr</i> M	Print 8 words of 'M'emory starting at <i>addr</i> .
M	Print 8 words <i>after</i> previous 'M' or 'N' command.
N	Print 8 words <i>before</i> previous 'M' or 'N' command.
<i>val</i> , <i>addr</i> /	Inspect/replace value at <i>addr</i> on CP1610 side.
<i>val</i> , /	Inspect/replace value at next address on CP1610 side.
<i>val</i> , <i>addr</i> ;	Inspect/replace value at <i>addr</i> on 6502 side.
<i>val</i> , ;	Inspect/replace value at next address on 6502 side.
<i>vvvv</i> , <i>addrlo</i> , <i>addrhi</i> S	'S'earch for 16-bit value in CP1610 address range (<i>addrlo</i> , <i>addrhi</i>).
<i>vv</i> , <i>addrlo</i> , <i>addrhi</i> S	'S'earch for 8-bit value ⁵ in CP1610 address range (<i>addrlo</i> , <i>addrhi</i>).
<i>addrlo</i> , <i>addrhi</i> V	'V'erify CP1610 download. Compute checksums over (<i>addrlo</i> , <i>addrhi</i>).
<i>addrlo</i> , <i>addrhi</i> , <i>len</i> U	'U'pload data from CP1610 (<i>addrlo</i> , <i>addrhi</i>) w/ <i>len</i> items per line.
<i>addr</i> \$B	Set 'B'reakpoint at <i>addr</i> on CP1610 side.
\$C	'C'lear all breakpoints on CP1610 side.
<i>addr</i> \$R	'R'emove breakpoint at <i>addr</i> on CP1610 side. ⁶
\$D	'D'isplay breakpoints on CP1610 side.
<i>dst</i> , <i>src</i> , <i>end</i> \$M	'M'ove (<i>src</i> , <i>end</i>) to (<i>dst</i> , ...) on CP1610 side.

⁵ Upper 8 bits *actively ignored* during this search. Number of digits in 'vv' determines which search is performed.

⁶ This command appears to be buggy. See details here: [\\$8951: CMD #\\$08: Clear breakpoint at address in R1](#).

The G, P and T commands fall into a command-proxy loop that allows the CP1610-side to interact with the serial port. To make a request from the CP1610 side, the code puts command numbers #10 and #11 in the command byte at \$05A3. Command #10 is “Receive byte from serial port for CP1610”, while command #11 is “Send byte to serial port.” See the code at [\\$E216](#) on the 6502 side for details.

CP1610 Download Record Format

The CP1610 download format bears a slight resemblance to Intel HEX format in its overall structure. It uses a different alphabet, however. Intel HEX uses normal hexadecimal values for its various fields. This monitor’s download format, on the other hand, uses ASCII characters 0x40 through 0x5F to represent 5-bit values. The 5-bit values themselves are extracted from the lower 5 bits of the ASCII code.

ASCII to Nickel Conversion Chart

ASCII	Char	5-Bit Value		ASCII	Char	5-Bit Value
0x40	@	0x00		0x50	P	0x10
0x41	A	0x01		0x51	Q	0x11
0x42	B	0x02		0x52	R	0x12
0x43	C	0x03		0x53	S	0x13
0x44	D	0x04		0x54	T	0x14
0x45	E	0x05		0x55	U	0x15
0x46	F	0x06		0x56	V	0x16
0x47	G	0x07		0x57	W	0x17
0x48	H	0x08		0x58	X	0x18
0x49	I	0x09		0x59	Y	0x19
0x4A	J	0x0A		0x5A	Z	0x1A
0x4B	K	0x0B		0x5B	[0x1B
0x4C	L	0x0C		0x5C	\	0x1C
0x4D	M	0x0D		0x5D]	0x1D
0x4E	N	0x0E		0x5E	^	0x1E
0x4F	O	0x0F		0x5F	_	0x1F

Download Record Fields

The CP1610 download format has 5 fields in the following order, terminated by a newline:

1. **Start of record.** This is a single colon ‘:’.
2. **Address.** This is encoded as 4 nickels (20 bits total), MSB-first, of which the lower 16 bits form the address.
3. **Data.** Each pair of characters comprises two nickels that form a decle. The first character corresponds to bits 9:5, while the second corresponds to bits 4:0.
4. **End of record.** This is a single semicolon ‘;’.
5. **Checksum.** This 16-bit checksum is computed by summing up all the ASCII values of the characters in the address and data fields, and then taking the 16-bit 2s complement of that sum. The ‘:’ and ‘;’ delimiters are not included in the sum. As with the address, this is sent as 4 nickels (20 bits total), MSB first, of which only the lower 16 bits are significant.

A typical download record looks like so:

: @T@@@DXB[A]BPB\BPCPB[BQBP@\BP@@@@@@@@@@@@@@@@@@@@GPG@FPF@EPE@DPD@BQBABSBT ; A[IC

Note that I’ve color-coded it to match the summary above, and inserted some whitespace to make the field boundaries clearer. The actual records have no whitespace other than the line-ending newline.

Note that the monitor does not seem to put an upper bound on the number of characters per record. However, common practice for file formats such as this is to keep the line length less than 80 characters. A record length of 32 decles results in a maximum line length of 74 characters.

Download Record Encoding Example

Consider the following 32 decles, which we intend to load at address \$5060.

```

17B 280 100 240 162 280 101 240
163 200 054 2BC 200 2B8 0F0 004
114 338 001 2B8 097 000 240 17F
001 2B8 050 000 240 180 2B8 17D

```

This block of decles encodes as follows:

Data	Bit Pattern as Nickels	Encoded	ASCII	New Checksum
5060	00000 10100 00011 00000	@TC@	40 54 43 40	0117
17B	01011 11011	K[4B 5B	01BD
280	10100 00000	T@	54 40	0251
100	01000 00000	H@	48 40	02D9
240	10010 00000	R@	52 40	036B

162	01011 00010	KB	4B 42	03F8
280	10100 00000	T@	54 40	048C
101	01000 00001	HA	48 41	0515
240	10010 00000	R@	52 40	05A7
163	01011 00011	KC	4B 43	0635
200	10000 00000	P@	50 40	06C5
054	00010 10100	BT	42 54	075B
2BC	10101 11100	U\	55 5C	080C
200	10000 00000	P@	50 40	089C
2B8	10101 11000	UX	55 58	0949
0F0	00111 10000	GP	47 50	09E0
004	00000 00100	@D	40 44	0A64
114	01000 10100	HT	48 54	0B00
338	11001 11000	YX	59 58	0BB1
001	00000 00001	@A	40 41	0C32
2B8	10101 11000	UX	55 58	0CDF
097	00100 10111	DW	44 57	0D7A
000	00000 00000	@@	40 40	0DFA
240	10010 00000	R@	52 40	0E8C
17F	01011 11111	K_	4B 5F	0F36
001	00000 00001	@A	40 41	0FB7
2B8	10101 11000	UX	55 58	1064
050	00010 10000	BP	42 50	10F6
000	00000 00000	@@	40 40	1176
240	10010 00000	R@	52 40	1208
180	01100 00000	L@	4C 40	1294
2B8	10101 11000	UX	55 58	1341
17D	01011 11101	K]	4B 5D	13E9

The 16-bit 2s complement of the checksum 13E9 is EC17. That encodes as follows:

EC16	00001 11011 00000 10111	A[@w	41 5B 40 57
------	-------------------------	------	-------------

Thus, the final encoded line looks like this:⁷

```
: @TC@ K[T@H@R@KBT@HAR@KCP@BTU\P@UXGP@DHTYX@AUXDW@@R@K_@AUXBP@@R@L@UXK] ; A[@W
```

6502 Download Record Format

The monitor also supports downloading code to the 6502 side. The monitor uses traditional [Intel HEX](#) (aka. IHEX8 or I8HEX) format for 6502 downloads. The monitor ignores the record type field, expecting all data-bearing records to be Data records. (EOF records have no data, and end up being ignored by the Monitor.)

The monitor distinguishes CP1610 data from 6502 data by the fact that Intel HEX data uses hexadecimal characters from the alphabet “0123456789ABCDEF”, while the CP1610 data uses ASCII characters 0x40 - 0x5F. That is, if the download routine sees a character in the range “0123456789”, it assumes the record is Intel HEX and switches to the 6502 record format.

If you look carefully, you’ll notice there’s a small overlap between the two alphabets. In particular, ASCII 0x41 - 0x46 are “ABCDEF”. In Intel HEX format, the first field provided is the line length field (in bytes of data), encoded in hex. As long as the line length is less than 160 bytes, the leading digit of the line will not be in the ambiguous range. Indeed, the typical line length for Intel HEX is 32 bytes per line. Therefore, the monitor’s heuristic should always work for real-world data.

See the [Wikipedia article for Intel HEX](#) for a full description of the Intel HEX format.

Loading 8-bit data in Intel HEX format explicitly clears bits 9:8 of the decs visible from the CP1610 side when loading into the shared dual-port RAM. ([See code at \\$E775.](#)) You cannot use the 6502 HEX format to load CP1610 code directly.

⁷ Whitespace added for readability. The whitespace must be omitted in the actual data sent to the monitor.

The Reverse Engineered Black Whale Monitor Code

The Black Whale monitor code consists of two pieces running in parallel: There is the 6502-side code which drives the serial port and implements the command parser/handler. There is also the CP1610-side code which accepts commands from the 6502-side to implement some of the required functionality. The following sections explore both halves. First, though, a quick KC hardware summary.

High Level Hardware Memory Map Summary

ROM Cartridge

The 6502 monitor code resides entirely on the U1 EPROM, with the valid monitor program residing at \$E000 - \$EC08.

The CP1610 monitor code resides entirely in the U3 EPROM, and is loaded into CP1610-visible memory during initialization. U3 is visible at \$8000 - \$8FFF in the 6502 address map. Its contents get loaded to locations \$8800 - \$8FFF on the CP1610 side during initialization.

The U2 ROM is the upper 4K of the 8K BASIC ROM, mapped at \$F000 - \$FFFF in the 6502 address map. Nothing in the Black Whale Monitor ROM appears to touch anything in this BASIC ROM. The only addresses that seem to get used in this ROM are the Reset/IRQ/NMI/BRK vectors from \$FFFA - \$FFFF.

Serial Cartridge

The serial board maps at \$B710 - \$B71F. Locations \$B710 - \$B713 are the R6551 ACIA. Locations \$B714 - \$B71F are aliases of \$B710 - \$B713.

Keyboard Component

This is just a high-level summary of the Keyboard Component memory map, necessary for understanding the Black Whale Monitor.

6502 Side

Range	Description
\$0000 - \$3FFF	Bits 7:0 of 16K x 10-bit dual-port RAM. Visible at \$8000 - \$BFFF on CP1610 side.
\$0000 - \$04FF	Reserved for Keyboard Component EXEC
\$0500 - \$3FFF	Available to user programs
\$4000 - \$40FF	I/O Space
\$4000 - \$4007	Peripheral inputs (tape, interrupt sources)

\$4020 - \$4027	Tape Control
\$4040 - \$4047	Various peripheral outputs (interrupt & display enables, tape data out)
\$4060	Keyboard read-back
\$4080	Clear tape interrupt pending
\$40A0	Clear SR1 interrupt pending
\$40Cx	TMS9927 CRTIC display controller
\$4100 - \$7FFF	Bits 9:8 of 16K x 10-bit dual-port RAM. Visible at \$8100 - \$BFFF on CP1610 side
\$8000 - \$8FFF	Black Whale ROM Cartridge: U3 ROM
\$A000 - \$AFFF	Unused
\$B000 - \$B7FF	I/O expansion space
\$B710 - \$B71F	Black Whale Serial Port
\$B800 - \$BFFF	CRTIC frame buffer + keyboard-private RAM
\$C000 - \$DFFF	Keyboard Component EXEC (6502 side)
\$E000 - \$EFFF	Black Whale ROM Cartridge: U1 ROM
\$F000 - \$FFFF	Black Whale ROM Cartridge: U2 ROM

CP1610 Side

Range	Description
\$0000 - \$003F	STIC Registers
\$0100 - \$01EF	8-bit Scratch RAM
\$0200 - \$02EF	BACKTAB
\$02F0 - \$035F	16-bit System RAM
\$1000 - \$1FFF	Intellivision EXEC
\$1041	Entry point for launching a game
\$5000 - \$6FFF	First 8K of standard cartridge image. (Provided by T-Card?)
\$7000 - \$7FFF	Keyboard Component EXEC (CP1610 side)
\$8000 - \$BFFF	16K x 10 dual port RAM
\$8000 - \$803F	STIC write sensitive
\$8000 - \$84FF	Reserved for KC EXEC
\$8500 - \$BFFF	Open for user programs
\$8800 - \$8FFF	Occupied by Black Whale Monitor Program (CP1610 code)

Black Whale Monitor, 6502 Side

Important 6502-side variables

Address(es)		Purpose
\$01		Scratch. Sometimes forms a checksum with \$11
\$05		Last byte from serial port.
\$06	\$08	Rightmost argument. \$08 is number of digits in argument.
\$0B	\$0E	Scratch / working space.
\$0F	\$11	Second argument from right. \$11 is number of digits in argument.
\$12	\$14	Third argument from right. \$14 is number of digits in argument.
\$05A3+\$4A53		Command word to CP1610-side. (\$85A3 on CP1610 side.)
\$05A4		8-bit argument exchange with CP1610-side. (R3 on CP1610)
\$05A5	\$05A6	16-bit argument exchange with CP1610-side. (R1 on CP1610, big endian)
\$05A7	\$05CA	Variable length argument buffer shared with CP1610-side. (@R4 on CP1610)
\$3FF9	\$3FFA	Handshake with CP1610-side during monitor initialization.

6502 \rightsquigarrow CP1610 command list

The 6502 side monitor issues commands to the CP1610. The CP1610 recognizes 15 commands, \$1 - \$F:

Command Number	Description
#\$00	Idle / No command.
#\$01	Copy R3 words from argument buffer to @R1.
#\$02	Display string in argument buffer; length in R3.
#\$03	Read a byte / decle from @R1 and write to argument buffer.
#\$04	Read a 16-bit value @R1 and write to argument buffer, little endian.
#\$05	Write 16-bit value from argument buffer to @R1, little endian.
#\$06	Resume execution at R1.
#\$07	Disable all breakpoints.
#\$08	Clear breakpoint at R1. This appears to be buggy.
#\$09	Set breakpoint at R1.
#\$0A	Copy breakpoint addresses to argument buffer, big endian.
#\$0B	Breakpoint sync: Spin in CMD dispatch loop. (Used for sync when halting.)

#\$0C	Copy saved CP1610 registers to arg buffer, big endian.
#\$0D	Set saved CP1610 register value. Reg # in R3, value in R1.
#\$0E	Single step instruction. If R1 != 0, run to instruction at location R1.
#\$0F	Resume execution at current saved PC.

Subroutine / branch target list

Address	Description
\$E000	Start of ROM / main entry point. Jumps to \$E004.
\$E003	ROM presence detect byte. Must be \$00.
\$E004	Actual entry ROM init entry point.
\$E04C	Delay 1 second and then stuff "X\r\n" into keyboard input FIFO.
\$E069	Menu entry for 'X' to copy to CP1610 side.
\$E08E	Main initialization entry point
\$E0BD	Send CR+LF and prompt; top of monitor command loop.
\$E0EB	Zero out partial argument list and re-enter command loop.
\$E0F3	Get command byte, decode and dispatch.
\$E150	Bad command: Send BEL, prompt, zero out args, and re-enter command loop.
\$E163	Dispatch for ':' (start download record)
\$E169	Dispatch for '\$' command: Branches to \$EB30. ⁸
\$E16C	Dispatch for 'S' command: Branches to \$E470.
\$E16F	Orphan? Branches to \$E4C1.
\$E172	Dispatch for 'V' command. Branches to \$E4C1.
\$E175	Dispatch for ';' command. Performs INC \$14, then branches to \$E3B0.
\$E177	Dispatch for '/' command. Branches to \$E3B0.
\$E17A	Dispatch for 'P' command. Branches to \$E310.
\$E17D	Dispatch for 'T' command. Branches to \$E31F.
\$E180	Dispatch for 'R', 'RR' commands. Branches to \$E25E.
\$E183	Dispatch for 'G' / 'GO' command. Branches to \$E1D0.
\$E186	Dispatch for 'Y' command. Branches to \$E1B4.
\$E189	Dispatch for 'U' command: Calls \$E85C, returns to command loop.
\$E18F	Dispatch for 'M' command: Branches to \$E32E.

⁸ Due to the limited reach of conditional branches on 6502, most command dispatches have a two-stage dispatch.

\$E192	Dispatch for 'N' command: Branches to \$E382.
\$E195	Try to receive 'ZERO' command. Goes to \$E58E if 'ZERO' received.
\$E1B4	Process 'Y' command: Resume execution at given address.
\$E1D0	Process 'G' / 'GO' command.
\$E216	Start CP1610 side running, and then poll for requests from it. Return to cmd loop on halt.
\$E25E	Process 'R' / 'RR' command.
\$E2A2	Show register information at current halt point.
\$E310	Process 'P' command ("Proceed"). Resume execution at current PC.
\$E31F	Process 'T' command. Run 't'o given address or single s't'ep.
\$E32E	Process 'M' command. Prints memory contents, stepping forward through memory.
\$E382	Process 'N' command. Prints memory contents, stepping backward through memory.
\$E3AA	Dead code? Alt entry for '/', ';' that increments address in \$07:\$06 first.
\$E3B0	Process '/' and ';' commands. Inspect/modify memory on CP1610 or 6502.
\$E470	Process 'S' command. Search for 8-bit or 16-bit values in CP1610 memory.
\$E4C1	Process 'V' command. Print out checksums for a range of memory.
\$E58E	Process 'ZERO' command: Zero out \$5000 - \$6FFF, \$9000 - \$9FFF on CP1610-side.
\$E5E4	Validate hex digit, decode and push into arg buffer at \$06 - \$07. C=1 on err, C=0 if ok.
\$E615	Text: "\024STARTING DOWNLOAD..."
\$E62A	Process a download record (':')
\$E666	Process checksum portion of a CP1610 download record (';')
\$E689	Report a bad record, drain serial until '\r' or record boundary.
\$E6A8	Get next declc into \$0E:\$0D from record.
\$E6C6	Get record address into \$0C:\$0B; or switch to Intel HEX format.
\$E6FB	Get next nickel from record being downloaded to us.
\$E71C	':' received during record download. Unwind stack and restart record parsing.
\$E723	';' received during record download. Unwind stack and validate checksum.
\$E72A	Error occurred during record download. Unwind stack and report "BAD RECORD."
\$E732	Download Intel HEX data for 6502 side.
\$E783	Read a byte of hexadecimal data from serial port (blocking).
\$E786	Same as \$E783, except first nibble is already in 'A'.
\$E799	Read a hex digit from the serial port (blocking).

\$E7B2	Print X:Y to the CRTC at \$B820 - \$B823.
\$E7D3	Convert upper nibble of A into a hexadecimal digit (ASCII).
\$E7D7	Convert lower nibble of A into a hexadecimal digit (ASCII).
\$E7E5	Print "RECORD OK . . . " to CRTC at \$B806.
\$E7F0	Print "BAD RECORD . . . " to CRTC at \$B806.
\$E7FB	Update on-screen error counter at \$B81D - \$B81E.
\$E81D	Various status strings
\$E85C	Process 'U' command. Upload data from CP1610 to the host.
\$E946	Send a single hex digit as one of "@ABCDEFGHJKLMNO" to the host (blocking).
\$E94E	Convert upper nibble of A into a hexadecimal digit (ASCII). ⁹
\$E952	Convert lower nibble of A into a hexadecimal digit (ASCII).
\$E960	Launch CP1610 side of debugger.
\$E9A3	Copy CP1610 code from \$8000 - \$8FFF (6502) to \$8800 - \$8FFF (CP1610).
\$E9CC	Set up addresses for copying CP1610 code to \$8800 on CP1610 side.
\$E9ED	Long delay, proportional to A squared.
\$EA01	Serial port Device Service Routine (DSR) record.
\$EA16	Serial port DSR initializer routine.
\$EA26	Serial port DSR interrupt handler.
\$EA8C	Set up serial port DSR as channel 2.
\$EAA4	Send CR+LF to host (blocking).
\$EAA9	Send LF to host (blocking).
\$EAAB	Send contents of A to host (blocking).
\$EABC	Receive byte from serial port (blocking).
\$EAC9	Test whether data is available on serial port. C=1 means "yes."
\$EAD2	Receive byte from serial port, with XON/XOFF/^C handling.
\$EAF1	Copy string to CRTC frame buffer at \$B806.
\$EB06	Print a 10-bit hex value in X:A to the host (blocking).
\$EB0E	Print an 8-bit hex value in A to the host (blocking).
\$EB17	Convert nibble to ASCII hex and send to host (blocking)
\$EB30	Process extended commands that start with '\$': \$M, \$B, \$C, \$R, \$D

⁹ The routines at \$E94E/\$E952 are *identical* to the routines at \$E7D3/\$E7D7.

\$EB4F	Process '\$B' command: Set breakpoint at 'addr'.
\$EB61	Process '\$C' command: Clear all active breakpoints.
\$EB69	Process '\$R' command: Remove breakpoint at 'addr'.
\$EB7B	Process '\$D' command: Display active breakpoints.
\$EBA9	Process '\$M' command: Memory copy.
\$EC00	Send command in A to CP1610 side; spin until CP1610 completes it.

\$E000: Launching the Monitor

Of all the code in the Black Whale Monitor, this is the code I understand the least. High level view: It appears this code installs a menu item named 'X', and then stuffs the sequence 'X\r\n' into the keyboard input FIFO to launch itself. Ultimately, we land at \$E08E to perform our actual initialization when this process completes.

Special note: Location \$E003 must be 0. The KC EXEC looks for this zero to detect the presence of an expansion ROM.

```

E000 4C 04 E0 JMP $E004

E003 .byte $00          ; Expansion ROM Present signature byte

;; Not really sure what this initializer does. It appears to be scanning
;; display memory starting at $B803, looking for an empty slot to register
;; that it's put a record of interest at $8500. It's weird that this is
;; in CRTC display memory though.

E004 8A          TXA
E005 48          PHA
E006 A2 B8      LDX #$B8    ; \_ Point ($96) at $B803, which is
E008 A9 03      LDA #$03    ; / part of CRTC display memory

E00A 85 97      STA $97     ; \_
E00C 86 96      STX $96     ; /
E00E A0 00      LDY #$00    ; \
E010 B1 96      LDA ($96),Y ; |
E012 85 98      STA $98     ; |
E014 C8          INY        ; |
E015 B1 96      LDA ($96),Y ; |
E017 85 99      STA $99     ; /
E019 05 98      ORA $98     ; \_ Was it $0000? If so, then $E030
E01B F0 13      BEQ $E030   ; /

E01D A4 FF      LDY $FF

E01F C8          INY
E020 B1 98      LDA ($98),Y

```

```

E022 C8      INY
E023 11 98   ORA ($98),Y
E025 D0 F8   BNE $E01F

E027 C8      INY
E028 B1 98   LDA ($98),Y
E02A C8      INY
E02B AA      TAX
E02C B1 98   LDA ($98),Y
E02E 10 DA   BPL $E00A

E030 A0 25   LDY #$25   ; \
E032 B9 68 E0 LDA $E068,Y ; | Copy 37 bytes of initializer data to
E035 99 FF 04 STA $04FF,Y ; |- $0500 - $0524. This is visible as $8500 -
E038 88      DEY      ; | $8524 on the CP1610 side.
E039 D0 F7   BNE $E032 ; /

E03B A9 73   LDA #$73   ; \_ Set two MSBs on JSR R4, $7353 at $8520 - 8522.
E03D 8D 22 45 STA $4522 ; / Only two LSBs observed. The reason this is
                        ; $73 is likely they used the "MSB-of-address"
                        ; operator in the original source.

E040 A9 00   LDA #$00
E042 91 96   STA ($96),Y
E044 A9 85   LDA #$85
E046 C8      INY
E047 91 96   STA ($96),Y
E049 68      PLA
E04A AA      TAX
E04B 60      RTS

```

\$E04C: Delay ~1 second and stuff "X\r\n" into keyboard input FIFO

```

E04C A2 00   LDX #$00   ; \
E04E A0 00   LDY #$00   ; |
E050 CA      DEX      ; |_ Delay a bit. (~1 second?)
E051 D0 FD   BNE $E050 ; |
E053 88      DEY      ; |
E054 D0 FA   BNE $E050 ; /

E056 20 B1 C6 JSR $C6B1 ; Get address of keyboard input FIFO
E059 A9 58   LDA #$58   ; \_ Stuff the letter 'X'
E05B 20 E0 DE JSR $DEE0 ; /
E05E A9 0D   LDA #$0D   ; \_ Stuff '\n'
E060 20 E0 DE JSR $DEE0 ; /
E063 A9 0A   LDA #$0A   ; \_ Stuff '\n'
E065 20 E0 DE JSR $DEE0 ; /
E068 60      RTS      ; Return

```

\$E069: Menu Item for 'X' copied to CP1610 side

```
E069 .byte $06, $85      ; \  Appears to be a pointer table of some sort for
E06B .byte $00, $00      ; |_ the CP1610 side. (These make sense as addr
E06D .byte $25, $85      ; |_ on that side.)
E06F .byte $20, $85      ; /
E071 .byte $58           ; 'X'
E072 .byte $00
E073 .byte $44, $4F, $57, $4E, $4C, $4F, $41, $44, $20, $4F ; DOWNLOAD O
E07D .byte $52, $20, $44, $45, $2D, $42, $55, $47, $2E, $2E ; R DE-BUG..
E087 .byte $2E, $00      ; .
E089 .byte $04, $70, $53 ; This becomes JSR R4, $7353, which is in the
                          ; rewind-and-eject-tape menu item. ??
E08C .byte $8E, $E0      ; Address of 6502 entry point below.
```

\$E08E: Main Initialization

One notable detail: The initialization routine sets up a standard Keyboard Component EXEC-style interrupt-driven Device Service Routine (DSR) for the serial port. That initialization routine sets up the serial port for nice, interrupt driven I/O, and even gives it a generous [82-byte input FIFO](#).

Then the code below disables interrupts, and the rest of the monitor resorts to polling on the serial port.

I suspect earlier versions of the Black Whale Monitor used the KC's infrastructure for serial I/O, but then ditched it when it didn't scale up to faster baud rates.

```
E08E A2 49    LDX #$49    ; 'I'
E090 8E 02 05 STX $0502  ; \
E093 CA      DEX        ; |_ ???
E094 8E 00 05 STX $0500  ; |
E097 8E 01 05 STX $0501  ; /
E09A 20 60 E9 JSR $E960  ; Launch CP1610 side
E09D 20 8C EA JSR $EA8C  ; Initialize serial port

E0A0 A9 00    LDA #$00    ; \
E0A2 8D A3 05 STA $05A3  ; |- Zero out command byte to CP1610 side.
E0A5 8D A3 45 STA $45A3  ; /

E0A8 85 04    STA $04    ; Clear ?flag?

E0AA A2 28    LDX #$28    ; \
E0AC A9 00    LDA #$00    ; |
E0AE 9D A2 05 STA $05A2,X ; |_ Clear argument buffer at $85A3-$85CA (CP1610)
E0B1 9D A2 45 STA $45A2,X ; |
E0B4 CA      DEX        ; |
E0B5 D0 F7    BNE $E0AE  ; /

E0B7 78      SEI        ; Disable interrupts
```

```

E0B8 A9 46 LDA #$46 ; 'F'
E0BA 8D 4D B9 STA $B94D ; Store to ?

```

\$E0BD: Top of monitor command loop: Send out CR+LF and '>' prompt.

```

E0BD 20 A4 EA JSR $EAA4 ; Send CR+LF out serial port (blocking)
E0C0 A9 3E LDA #$3E ; \__ Send '>' out serial port (blocking)
E0C2 20 AB EA JSR $EAAB ; /

E0C5 A9 00 LDA #$00 ; \
E0C7 85 0F STA $0F ; |
E0C9 85 10 STA $10 ; |
E0CB 85 11 STA $11 ; | - Zero $0F through $14
E0CD 85 12 STA $12 ; |
E0CF 85 13 STA $13 ; |
E0D1 85 14 STA $14 ; /

;; Arguments are stored in this order:
;;
;; $13:$12, $10:$0F, $07:$06
;;
;; $14, $11, and $08 store the number of digits in each argument,
;; to indicate whether the argument was provided, and if so, how
;; long it was. (Length matters for some commands.)
;;
;; Arguments are separated by commas or less-than signs.

;; Shift all the arguments over by one:
E0D3 A5 0F LDA $0F
E0D5 85 12 STA $12
E0D7 A5 10 LDA $10
E0D9 85 13 STA $13
E0DB A5 11 LDA $11
E0DD 85 14 STA $14

E0DF A5 06 LDA $06
E0E1 85 0F STA $0F
E0E3 A5 07 LDA $07
E0E5 85 10 STA $10
E0E7 A5 08 LDA $08
E0E9 85 11 STA $11

```

\$E0EB: Zero out rightmost argument and re-enter command loop

```

;; Zero out the rightmost (currently incoming) argument:
E0EB A9 00 LDA #$00 ;
E0ED 85 08 STA $08 ;
E0EF 85 07 STA $07 ;
E0F1 85 06 STA $06 ;

```

\$E0F3: Receive command byte, decode and dispatch

```
E0F3 20 BC EA JSR $EABC ; Wait for byte from serial port
E0F6 C9 3A CMP #$3A ; \_ ':'
E0F8 F0 69 BEQ $E163 ; /
E0FA C9 58 CMP #$58 ; \_ 'X': Zero out partial incoming argument
E0FC F0 ED BEQ $E0EB ; / and re-enter the receive loop.
E0FE C9 0D CMP #$0D ; CR
E100 F0 BB BEQ $E0BD
E102 C9 20 CMP #$20 ; \_ Non-printing: Zero out the partial incoming
E104 90 E5 BCC $E0EB ; / argument and re-enter the receive loop.
E106 48 PHA
E107 20 AB EA JSR $EAAB ; Echo it back
E10A 68 PLA
E10B C9 2C CMP #$2C ; ',': Shift all the arguments over.
E10D F0 C4 BEQ $E0D3
E10F C9 3C CMP #$3C ; '<'
E111 F0 C0 BEQ $E0D3
E113 C9 24 CMP #$24 ; '$'
E115 F0 52 BEQ $E169
E117 C9 56 CMP #$56 ; 'V'
E119 F0 57 BEQ $E172
E11B C9 50 CMP #$50 ; 'P'
E11D F0 5B BEQ $E17A
E11F C9 54 CMP #$54 ; 'T'
E121 F0 5A BEQ $E17D
E123 C9 53 CMP #$53 ; 'S'
E125 F0 45 BEQ $E16C
E127 C9 47 CMP #$47 ; 'G'
E129 F0 58 BEQ $E183
E12B C9 59 CMP #$59 ; 'Y'
E12D F0 57 BEQ $E186
E12F C9 4D CMP #$4D ; 'M'
E131 F0 5C BEQ $E18F
E133 C9 52 CMP #$52 ; 'R'
E135 F0 49 BEQ $E180
E137 C9 55 CMP #$55 ; 'U'
E139 F0 4E BEQ $E189
E13B C9 2F CMP #$2F ; '/'
E13D F0 38 BEQ $E177
E13F C9 3B CMP #$3B ; ';'
E141 F0 32 BEQ $E175
E143 C9 4E CMP #$4E ; 'N'
E145 F0 4B BEQ $E192
E147 C9 5A CMP #$5A ; 'Z'
E149 F0 4A BEQ $E195
E14B 20 E4 E5 JSR $E5E4 ; Is this a hex character we can accumulate?
E14E 90 A3 BCC $E0F3 ; Yes: Continue.
```

\$E150: Bad Command: Send BEL, '>', CR+LF and start over

```
E150 20 A4 EA JSR $EAA4 ; Send CR+LF
E153 A9 07 LDA #$07 ; \_ send a BEL
E155 20 AB EA JSR $EAAB ; /
E158 A9 3F LDA #$3F ; \_ Send a '>'
E15A 20 AB EA JSR $EAAB ; /
E15D 20 A4 EA JSR $EAA4 ; Send a CR+LF
E160 4C EB E0 JMP $E0EB
```

\$E163: Second-level dispatches for monitor commands

Note that the dispatch for ';' merely increments \$14 before falling into the dispatch for '/'. This creates a phantom 3rd argument that the '/' code uses to distinguish ';' from '/'.

Also, the jump at \$E16F doesn't seem to be called by anything, and seems to be an orphan.

```
E163 20 2A E6 JSR $E62A ; Dispatch for ':'
E166 4C BD E0 JMP $E0BD ; Return to command loop

E169 4C 30 EB JMP $EB30 ; Dispatch for '$'
E16C 4C 70 E4 JMP $E470 ; Dispatch for 'S'
E16F 4C C1 E4 JMP $E4C1 ; Orphan?
E172 4C C1 E4 JMP $E4C1 ; Dispatch for 'V'

;; The ';' and '/' commands are closely related:
E175 E6 14 INC $14 ; ';' branches here, falls thru to JMP
E177 4C B0 E3 JMP $E3B0 ; Dispatch for both ';' and '/'.

E17A 4C 10 E3 JMP $E310 ; Dispatch for 'P'
E17D 4C 1F E3 JMP $E31F ; Dispatch for 'T'
E180 4C 5E E2 JMP $E25E ; Dispatch for 'R'
E183 4C D0 E1 JMP $E1D0 ; Dispatch for 'G'
E186 4C B4 E1 JMP $E1B4 ; Dispatch for 'Y'

E189 20 5C E8 JSR $E85C ; Dispatch for 'U'
E18C 4C BD E0 JMP $E0BD ; Return to command loop

E18F 4C 2E E3 JMP $E32E ; Dispatch for 'M'
E192 4C 82 E3 JMP $E382 ; Dispatch for 'N'
```

\$E195: Receive the "ZERO" command

```
;; Get four more bytes. Expect "ERO\r".
;; Since we come here after a 'Z', the whole command is "ZERO".
E195 20 BC EA JSR $EABC ; \
E198 C9 45 CMP #$45 ; |- 'E'? No: Error out.
E19A D0 B4 BNE $E150 ; /
```

```

E19C 20 BC EA JSR $EABC ; \
E19F C9 52 CMP #$52 ; |- 'R'? No: Error out.
E1A1 D0 AD BNE $E150 ; /
E1A3 20 BC EA JSR $EABC ; \
E1A6 C9 4F CMP #$4F ; |- 'O'? No: Error out.
E1A8 D0 A6 BNE $E150 ; /
E1AA 20 BC EA JSR $EABC ; \
E1AD C9 0D CMP #$0D ; |- '\r'? No: Error out.
E1AF D0 9F BNE $E150 ; /
E1B1 4C 8E E5 JMP $E58E ; ZERO OUT THE WORLD!

```

\$E1B4: Process the 'Y' command: Resume CP1610 execution at given address

Unlike the 'G' command, this starts the CP1610, but does not bridge serial port access between the CP1610 side and the rest of the world. It just starts the CP1610 side and lets it run free.

```

E1B4 20 BC EA JSR $EABC ; \
E1B7 C9 0D CMP #$0D ; |- Get a char. If '\r' continue.
E1B9 F0 03 BEQ $E1BE ; /
E1BB 4C 50 E1 JMP $E150 ; Not '\r': BEL and error out.

E1BE A5 07 LDA $07 ; \
E1C0 8D A5 05 STA $05A5 ; |- Load argument address
E1C3 A5 06 LDA $06 ; |
E1C5 8D A6 05 STA $05A6 ; /
E1C8 A9 06 LDA #$06 ; \_ Issue CMD #$06: Run at R1
E1CA 8D A3 05 STA $05A3 ; /
E1CD 4C BD E0 JMP $E0BD ; Go back to monitor loop.

```

\$E1D0: Dispatch to 'G' vs. 'GO' command

```

E1D0 20 BC EA JSR $EABC ; Get a char.
E1D3 C9 0D CMP #$0D ; \_ Is it '\r'?
E1D5 F0 0D BEQ $E1E4 ; / Yes: E1E4

E1D7 C9 4F CMP #$4F ; \_ 'O'?
E1D9 F0 06 BEQ $E1E1 ; / Yes: Jump to 6502 address in first arg.
E1DB 20 AB EA JSR $EAAB ; Echo character back to user.
E1DE 4C 50 E1 JMP $E150 ; Error out.

```

\$E1E1: Process 'GO' command

```

E1E1 6C 06 00 JMP ($0006) ; Jump to user-provided addr on 6502 side.

```

\$E1E4: Process 'G' command

This is more than just launching the CP1610 side. It blanks the CRTC memory, launches the CP1610 side, and then drops into an active polling loop to bridge the serial port over to requests from the CP1610 side.

The 6502 side stays in a tight monitoring loop watching for breakpoints or other requests from the CP1610 side.

```

E1E4 A9 10 LDA #$10 ; \
E1E6 8D A5 05 STA $05A5 ; |_ Address $1041... the address to resume
E1E9 A9 41 LDA #$41 ; | booting the EXEC after probing $7000, $4800.
E1EB 8D A6 05 STA $05A6 ; /
E1EE A5 08 LDA $08 ; \ Hex argument provided?
E1F0 F0 0A BEQ $E1FC ; / No: goto E1FC
E1F2 A5 07 LDA $07 ; \
E1F4 8D A5 05 STA $05A5 ; |_ Override $1041 with the address provided.
E1F7 A5 06 LDA $06 ; |
E1F9 8D A6 05 STA $05A6 ; /
E1FC A9 B8 LDA #$B8 ; \
E1FE 85 07 STA $07 ; |_ Replace arg buffer with $B800.
E200 A9 00 LDA #$00 ; |
E202 85 06 STA $06 ; /
E204 A8 TAY ; \
E205 A9 80 LDA #$80 ; |
E207 91 06 STA ($06),Y ; |
E209 C8 INY ; |
E20A D0 FB BNE $E207 ; |- Fill $B800 - $BFFF with #$80.
E20C E6 07 INC $07 ; |
E20E A5 07 LDA $07 ; |
E210 C9 C0 CMP #$C0 ; |
E212 D0 F1 BNE $E205 ; /
E214 A9 06 LDA #$06 ; CMD #$06: Have CP1610 side jump to given addr.

```

; This falls through to the code at \$E216.

\$E216: Run code on CP1610 side, and poll for requests from it, until CP1610 halts

This code is used by the G, P and T commands to start the CP1610-side running. The G command passes in command #\$6 in 'A', while the P code passes in command #\$F, and T passes in command #\$E. In all three cases, the code falls into a loop which accepts requests from the CP1610 side to communicate over serial, bridging the serial link to code running on the Master Component.

Once the CP1610 side halts, control returns to the debugger command loop.

```

E216 8D A3 05 STA $05A3 ; Issue command in 'A' to CP1610 side.

E219 20 C9 EA JSR $EAC9 ; Is there data on the serial port?
E21C 66 01 ROR $01 ; Deposit data-available flag in $01
E21E 10 0C BPL $E22C ; No: Go to E22C.
E220 20 BC EA JSR $EABC ; Get char from serial port
E223 C9 1B CMP #$1B ; \ ESC? No: Ignore it.
E225 D0 05 BNE $E22C ; /
E227 A9 36 LDA #$36 ; \ ESC => Interrupt execution on the CP1610 side.
E229 8D A3 05 STA $05A3 ; / (Checked for in monitor ISR.)

```

```

E22C AD A3 05 LDA $05A3 ; Req from CP1610 => 6502 side?
E22F C9 10 CMP #$10 ; \_ Request to receive byte from serial port
E231 F0 0E BEQ $E241 ; /
E233 C9 11 CMP #$11 ; \_ Request to send byte @$05A5 out serial port.
E235 F0 1E BEQ $E255 ; / (Print request from CP1610 side.)
E237 C9 0B CMP #$0B ; \_ Has it halted yet? (CP1610 halts from ISR)
E239 D0 DE BNE $E219 ; / No? Loop back to getting char from serial port
E23B 20 A2 E2 JSR $E2A2 ; Yes: Show details about where we halted.
E23E 4C BD E0 JMP $E0BD ; Back to command loop

E241 A5 01 LDA $01 ; \_ Data available flag
E243 8D A4 05 STA $05A4 ; /
E246 10 05 BPL $E24D ; If unset, don't copy the data over (it's stale)
E248 A5 05 LDA $05 ; \_ Copy the data over
E24A 8D A5 05 STA $05A5 ; /
E24D A9 00 LDA #$00 ; \
E24F 8D A3 05 STA $05A3 ; |- CMD complete, zero it out and resume polling.
E252 4C 19 E2 JMP $E219 ; /

E255 AD A5 05 LDA $05A5 ; Get byte to send
E258 20 AB EA JSR $EAAB ; Send it
E25B 4C 4D E2 JMP $E24D ; CMD complete

```

\$E25E: Process R / RR Command

The *r*, *vvvvR* command updates register *Rr* to value *vvvv* on the CP1610 side. In contrast, *RR* just prints the current set of register values from the CP1610 side.

```

E25E 20 BC EA JSR $EABC ; \
E261 48 PHA ; |- Get byte from serial port and echo it back
E262 20 AB EA JSR $EAAB ; |
E265 68 PLA ; /
E266 C9 0D CMP #$0D ; \_ '\r'? Command was "R"
E268 F0 07 BEQ $E271 ; /
E26A C9 52 CMP #$52 ; \_ "RR": Just print out current register state.
E26C F0 2E BEQ $E29C ; /
E26E 4C 50 E1 JMP $E150 ; Neither: Error out and loop for new command.

;; 'R' command
E271 A9 0C LDA #$0C ; \
E273 8D A3 05 STA $05A3 ; |- Ask CP1610 side for the saved register state.
E276 AD A3 05 LDA $05A3 ; | (Not sure we do anything with it though.)
E279 D0 FB BNE $E276 ; /
E27B A5 10 LDA $10 ; \
E27D D0 EF BNE $E26E ; |
E27F A5 0F LDA $0F ; |- Error out if R's second argument is >= 9
E281 C9 09 CMP #$09 ; |
E283 B0 E9 BCS $E26E ; /
E285 8D A4 05 STA $05A4 ; Write register number to manipulate
E288 A5 06 LDA $06 ; \
E28A 8D A6 05 STA $05A6 ; |- Write value to update to.

```

```

E28D A5 07 LDA $07 ; |
E28F 8D A5 05 STA $05A5 ; /
E292 A9 0D LDA #$0D ; \
E294 8D A3 05 STA $05A3 ; |_ Ask CP1610 side to update its saved reg.
E297 AD A3 05 LDA $05A3 ; |
E29A D0 FB BNE $E297 ; /

;; "RR" command (also, tail of "R" command)
E29C 20 A2 E2 JSR $E2A2 ; Show details of the previous state.
E29F 4C BD E0 JMP $E0BD ; Loop waiting for next command

```

\$E2A2: Show details of where we halted

This is called from multiple places, including RR, and when G detects we've halted. It prints a record such as the following out to the host via the serial port:

```
S R0=xxxx R1=xxxx R2=xxxx R3=xxxx R4=xxxx R5=xxxx R6=xxxx/xxxx
```

The code:

```

E2A2 A9 00 LDA #$00 ; \_ Initialize loop counter
E2A4 85 11 STA $11 ; /
E2A6 A9 0C LDA #$0C ; \
E2A8 8D A3 05 STA $05A3 ; |_ Ask CP1610 for its saved register state.
E2AB AD A3 05 LDA $05A3 ; | (CMD #$0C)
E2AE D0 FB BNE $E2AB ; /
E2B0 20 A4 EA JSR $EAA4 ; CR+LF out serial port
E2B3 A9 53 LDA #$53 ; \_ 'S' out serial port
E2B5 20 AB EA JSR $EAAB ; /
E2B8 4C CE E2 JMP $E2CE ; Jump to loop test

E2BB A9 20 LDA #$20 ; \
E2BD 20 AB EA JSR $EAAB ; |_ " R" out serial port
E2C0 A9 52 LDA #$52 ; |
E2C2 20 AB EA JSR $EAAB ; /
E2C5 A6 11 LDX $11 ; \
E2C7 CA DEX ; | Decimalize and print register number out
E2C8 8A TXA ; | - serial port. (Hi bit set on ASCII, why?)
E2C9 09 B0 ORA #$B0 ; |
E2CB 20 AB EA JSR $EAAB ; /
E2CE A9 3D LDA #$3D ; \_ '=' out serial port
E2D0 20 AB EA JSR $EAAB ; /
E2D3 A5 11 LDA $11 ; \
E2D5 0A ASL A ; | - Index to selected register (2*A)
E2D6 A8 TAY ; /
E2D7 48 PHA ; Save index on stack
E2D8 B9 A7 05 LDA $05A7,Y ; Get MSB
E2DB 8D A5 05 STA $05A5 ; Remember MSB
E2DE 20 0E EB JSR $EB0E ; MSB hex out serial port
E2E1 68 PLA ; \_ Restore index

```

```

E2E2 A8      TAY          ; /
E2E3 B9 A8 05 LDA $05A8,Y ; Get LSB
E2E6 8D A6 05 STA $05A6   ; Remember LSB
E2E9 20 0E EB JSR $EB0E   ; LSB hex out serial port

E2EC A5 11      LDA $11    ; \
E2EE E6 11      INC $11    ; |_ Iterate over R0..R6.
E2F0 C9 08      CMP #$08   ; | (Loop counter goes 1..7, exits on 8)
E2F2 D0 C7      BNE $E2BB  ; /

E2F4 A9 04      LDA #$04   ; \_ Read 16-bit value from CP1610 side. This
E2F6 8D A3 05 STA $05A3   ; / should be @ saved R6 value. (Interrupted PC?)
E2F9 A9 2F      LDA #$2F   ; \_ '/' out serial port
E2FB 20 AB EA JSR $EAAB   ; /
E2FE AD A3 05 LDA $05A3   ; \_ Wait until CMD complete
E301 D0 FB      BNE $E2FE  ; /
E303 AD A8 05 LDA $05A8   ; \
E306 20 0E EB JSR $EB0E   ; |_ Output interrupted PC out serial port
E309 AD A7 05 LDA $05A7   ; |
E30C 20 0E EB JSR $EB0E   ; /
E30F 60          RTS          ; Done.

```

\$E310: Process 'P' command: Proceed with execution at current PC

```

E310 20 BC EA JSR $EABC   ; Get char from serial
E313 C9 0D      CMP #$0D   ; \_ Not '\r'? Error out.
E315 D0 05      BNE $E31C  ; /
E317 A9 0F      LDA #$0F   ; CMD #$0F: Resume execution
E319 4C 16 E2 JMP $E216   ; Issue CMD and go to polling loop
E31C 4C 50 E1 JMP $E150   ; Error out.

```

\$E31F: Process 'T' command: Run 'T'o address, or single s'T'ep

```

E31F A5 07      LDA $07    ; \
E321 8D A5 05 STA $05A5   ; |_ Address to run to; if 0, single step
E324 A5 06      LDA $06    ; |
E326 8D A6 05 STA $05A6   ; /
E329 A9 0E      LDA #$0E   ; \_ CMD #$0E: Single step / run to
E32B 4C 16 E2 JMP $E216   ; / Issue CMD and go to polling loop

```

\$E32E: Process 'M' command: Print 8 memory values from CP1610 side

If no argument is provided, the M command starts after a previous M command left off. (At least, that appears to be the intention.) The closely related N command works similarly, but steps backward through memory. The N command implementation immediately follows this code, and the two should be considered together.

```

E32E 20 34 E3 JSR $E334   ; Body of command below.
E331 4C D3 E0 JMP $E0D3   ; Get next command

```

```

E334 A5 08 LDA $08 ; Any arguments?
E336 F0 0A BEQ $E342 ; No: Use last address
E338 A5 07 LDA $07 ; \
E33A 8D A5 05 STA $05A5 ; |_ Copy over argument
E33D A5 06 LDA $06 ; |
E33F 8D A6 05 STA $05A6 ; /
E342 20 A4 EA JSR $EAA4 ; CR+LF out
E345 AD A5 05 LDA $05A5 ; \
E348 20 0E EB JSR $EB0E ; |_ Print address in hex out serial port
E34B AD A6 05 LDA $05A6 ; |
E34E 20 0E EB JSR $EB0E ; /
E351 A9 3A LDA #$3A ; \ ':'
E353 20 AB EA JSR $EAAB ; /
E356 A9 08 LDA #$08 ; \ 8 values per line
E358 85 06 STA $06 ; /

E35A A9 04 LDA #$04 ; \
E35C 8D A3 05 STA $05A3 ; |_ Ask CP1610 to read us the value
E35F AD A3 05 LDA $05A3 ; |
E362 D0 FB BNE $E35F ; /
E364 AD A8 05 LDA $05A8 ; \
E367 20 0E EB JSR $EB0E ; |_ Print value out serial port
E36A AD A7 05 LDA $05A7 ; |
E36D 20 0E EB JSR $EB0E ; /
E370 A9 20 LDA #$20 ; \ ' ' out serial port
E372 20 AB EA JSR $EAAB ; /
E375 EE A6 05 INC $05A6 ; \
E378 D0 03 BNE $E37D ; |- Next address
E37A EE A5 05 INC $05A5 ; /
E37D C6 06 DEC $06 ; \ Loop over all 8
E37F D0 D9 BNE $E35A ; /
E381 60 RTS

```

\$E382: Process 'N' command: Like 'M', but steps backward through memory

```

E382 AD A6 05 LDA $05A6 ; \
E385 38 SEC ; |
E386 E9 08 SBC #$08 ; |
E388 8D A6 05 STA $05A6 ; |_ Rewind by 8
E38B AD A5 05 LDA $05A5 ; |
E38E E9 00 SBC #$00 ; |
E390 8D A5 05 STA $05A5 ; /
E393 20 42 E3 JSR $E342 ; Print out the 8 values via serial
E396 AD A6 05 LDA $05A6 ; \
E399 38 SEC ; |
E39A E9 08 SBC #$08 ; |
E39C 8D A6 05 STA $05A6 ; |_ Rewind by 8 again
E39F AD A5 05 LDA $05A5 ; |
E3A2 E9 00 SBC #$00 ; |
E3A4 8D A5 05 STA $05A5 ; /
E3A7 4C D3 E0 JMP $E0D3 ; Return to command loop

```

\$E3AA: Dead code

This code appears to be dead. It falls into the '/' and ';' processing code, and appears to have been meant for a "process next address" loop. However, nothing I've found actually comes to this address.

```
E3AA E6 06    INC $06
E3AC D0 02    BNE $E3B0
E3AE E6 07    INC $07
```

\$E3B0: Process '/' and ';' commands: Inspect and replace values in memory

The / and ; commands inspect and replace values in memory. The / command operates on the CP1610 side, while the ; command operates on the 6502 side.

Both commands take a value followed by an optional address. The syntax is *vvvv,addr/* or *vvvv,addr;* where *vvvv* is the value, and *addr* is the address. If the address is omitted, the command will use the previous address. The comma is still required: e.g. *vvvv,/*.

```
                ; Prepare CMD record
E3B0 A5 08    LDA $08      ; \_ Last argument non-empty?
E3B2 F0 0A    BEQ $E3BE    ; / No: Use previous address.
E3B4 A5 07    LDA $07      ; \
E3B6 8D A5 05 STA $05A5    ; |_ Yes: Copy the address over.
E3B9 A5 06    LDA $06      ; |
E3BB 8D A6 05 STA $05A6    ; /
E3BE A9 01    LDA #$01     ; \_ One word
E3C0 8D A4 05 STA $05A4    ; /
E3C3 A5 14    LDA $14      ; \_ $14 = 1 means ';' (6502)
E3C5 F0 14    BEQ $E3DB    ; / $14 = 0 means '/' (CP1610)

                ; 6502 handling
E3C7 AD A6 05 LDA $05A6    ; \
E3CA 85 06    STA $06      ; |_ Copy address back to rightmost arg
E3CC AD A5 05 LDA $05A5    ; | out of CMD record.
E3CF 85 07    STA $07      ; /
E3D1 A0 00    LDY #$00
E3D3 B1 06    LDA ($06),Y ; \_ Read a byte from the 6502 side @ address
E3D5 8D A7 05 STA $05A7    ; /
E3D8 4C EB E3 JMP $E3EB    ; Display the byte

                ; CP1610 handling
E3DB A9 04    LDA #$04     ; \
E3DD 8D A3 05 STA $05A3    ; |_ Ask CP1610 to read the word for us (CMD #4)
E3E0 AD A3 05 LDA $05A3    ; |
E3E3 D0 FB    BNE $E3E0    ; /

E3E5 AD A8 05 LDA $05A8    ; \_ Print out MSB of value from CP1610 to serial
E3E8 20 0E EB JSR $EB0E    ; /

E3EB AD A7 05 LDA $05A7    ; \_ Print out LSB of value to serial.
E3EE 20 0E EB JSR $EB0E    ; / (Common code for CP1610 and 6502 paths.)
```

```

E3F1 A9 20 LDA #$20 ; \_ ' ' out serial
E3F3 20 AB EA JSR $EAAB ; /
E3F6 A9 00 LDA #$00 ; \
E3F8 85 06 STA $06 ; |_ Zero out the argument so we reuse the address
E3FA 85 07 STA $07 ; |
E3FC 85 08 STA $08 ; /
E3FE 20 BC EA JSR $EABC ; Block waiting for char
E401 C9 0A CMP #$0A ; \_ ^J : ???
E403 F0 07 BEQ $E40C ; /
E405 20 AB EA JSR $EAAB ; Echo it back
E408 C9 0D CMP #$0D ; \_ '\r'?
E40A D0 56 BNE $E462 ; / No: Try to accumulate hex digits for a
E40C A5 08 LDA $08 ; replacement
E40E F0 2D BEQ $E43D ; If the argument is empty, skip writing the update

E410 A5 06 LDA $06 ; \
E412 8D A7 05 STA $05A7 ; |_ prepare replacement value for CP1610 side
E415 A5 07 LDA $07 ; |
E417 8D A8 05 STA $05A8 ; /
E41A A5 14 LDA $14 ; \_ Branch to CP1610 update code if fewer
E41C F0 15 BEQ $E433 ; / than 3 args.
E41E A5 06 LDA $06 ; \_ Get and save byte to write
E420 48 PHA ; /
E421 AD A5 05 LDA $05A5 ; \
E424 85 07 STA $07 ; |_ Copy 6502 address to $07:$06
E426 AD A6 05 LDA $05A6 ; |
E429 85 06 STA $06 ; /
E42B 68 PLA ; \
E42C A0 00 LDY #$00 ; |_ Restore byte to write and write it.
E42E 91 06 STA ($06),Y ; /
E430 4C 3D E4 JMP $E43D ; Move to next location.

E433 A9 05 LDA #$05 ; \
E435 8D A3 05 STA $05A3 ; |_ Send CMD #5 to update word on CP1610 side,
E438 AD A3 05 LDA $05A3 ; | and wait for it to complete.
E43B D0 FB BNE $E438 ; /

E43D A5 05 LDA $05 ; \
E43F C9 0D CMP #$0D ; |_ Was last byte of input '\r'?
E441 F0 27 BEQ $E46A ; / Yes: Go back to command loop

E443 20 A4 EA JSR $EAA4 ; CR+LF to serial output

;; Moving on to the next address
E446 EE A6 05 INC $05A6 ; \
E449 D0 03 BNE $E44E ; |_ Increment target address
E44B EE A5 05 INC $05A5 ; /

E44E AD A5 05 LDA $05A5 ; \
E451 20 0E EB JSR $EB0E ; |
E454 AD A6 05 LDA $05A6 ; |_ Print new target address followed by ':'
E457 20 0E EB JSR $EB0E ; |

```

```

E45A A9 2F LDA #$2F ; |
E45C 20 AB EA JSR $EAAB ; /
E45F 4C BE E3 JMP $E3BE ; Do this all again for the next address.

;; Try to accumulate hex digits for a replacement
E462 20 E4 E5 JSR $E5E4 ; Interpret char as hex digit. C=1 means error
E465 B0 06 BCS $E46D ; C=1: Error out
E467 4C FE E3 JMP $E3FE ; Go back to serial polling loop.

E46A 4C BD E0 JMP $E0BD ; Return to command loop

E46D 4C 50 E1 JMP $E150 ; Error out

```

\$E470: Process the 'S' command: Search for 8 or 16-bit values in CP1610 memory

This command searches for 8-bit or 16-bit values in CP1610 memory. It determines whether to search for 8-bit or 16-bit values by looking at the number of digits provided in the value. If it's fewer than 3 digits, it performs an 8-bit search. Otherwise it performs a 16-bit search. The 8-bit search explicitly ignores the upper 8 bits of the value.

```

E470 A5 14 LDA $14 ; \ 'S' requires 3 arguments.
E472 F0 F9 BEQ $E46D ; / Error out.
E474 A5 0F LDA $0F ; \
E476 8D A6 05 STA $05A6 ; |_ Copy 'addrlo' argument over
E479 A5 10 LDA $10 ; |
E47B 8D A5 05 STA $05A5 ; /
E47E A9 04 LDA #$04 ; \
E480 8D A3 05 STA $05A3 ; |_ Read the location from CP1610 side (CMD #4)
E483 AD A3 05 LDA $05A3 ; |
E486 D0 FB BNE $E483 ; /
E488 AD A7 05 LDA $05A7 ; \
E48B C5 12 CMP $12 ; |- Is it equal to the third argument?
E48D D0 1A BNE $E4A9 ; / No: E4A9
E48F A5 14 LDA $14 ; \
E491 C9 03 CMP #$03 ; |- Fewer than 3 digits on vv: treat as 8-bit
E493 90 07 BCC $E49C ; /
E495 AD A8 05 LDA $05A8 ; \ Check upper byte if 16-bit
E498 C5 13 CMP $13 ; /
E49A D0 0D BNE $E4A9 ; Not equal: E4A9
;; Found the value being searched for.
E49C 20 A4 EA JSR $EAA4 ; CR+LF out to serial
E49F A5 10 LDA $10 ; \
E4A1 20 0E EB JSR $EB0E ; |_ Print the address.
E4A4 A5 0F LDA $0F ; |
E4A6 20 0E EB JSR $EB0E ; /
;; Loop check
E4A9 A5 0F LDA $0F ; \
E4AB C5 06 CMP $06 ; |
E4AD D0 09 BNE $E4B8 ; |_ Have we reached the end of the range (addrhi)?
E4AF A5 10 LDA $10 ; | No: move to next address
E4B1 C5 07 CMP $07 ; |
E4B3 D0 03 BNE $E4B8 ; /

```



```
E4B5 4C BD E0 JMP $E0BD ; Yes: Go back to command loop
```

```
E4B8 E6 0F INC $0F ; \  
E4BA D0 02 BNE $E4BE ; |_ Increment address and do it again.  
E4BC E6 10 INC $10 ; |  
E4BE 4C 74 E4 JMP $E474 ; /
```

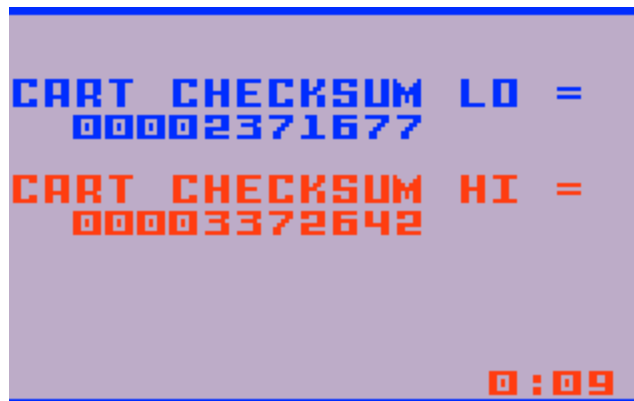
\$E4C1: Process the 'V' command: Verify a region by computing checksums

This actually computes two kinds of checksums: A 16-bit checksum of 8-bit values, similar to what you'd find written on an EPROM, and a 32-bit checksum of 16-bit values, similar to that output by MTE-201.

The 16-bit checksums on 8-bit values are computed separately on the lower and upper bytes of each 16-bit word and are reported in hexadecimal. This matches common practice for checksums on EPROMs.

The 32-bit checksum on 16-bit values is a simple sum of the 16-bit values. The resulting 32-bit checksum is output as an *octal* value, with only the lower 27 bits reported. I believe this is the same checksum style used by MTE-201.

The image below illustrates MTE-201's octal checksum. It reports 11 digits, so a full 33-digit checksum; however the upper 4 digits are all 0. LO/HI refers to \$5xxx and \$7xxx, not upper/lower byte.



```
E4C1 20 A4 EA JSR $EAA4 ; CR+LF out to serial
```

```
E4C4 A9 00 LDA #$00  
E4C6 85 0C STA $0C ; \_ Zero out 16-bit checksum of upper bytes  
E4C8 85 0B STA $0B ; /  
E4CA 85 0E STA $0E ; \_ Zero out 16-bit checksum of lower bytes  
E4CC 85 0D STA $0D ; /  
E4CE 85 12 STA $12 ; \  
E4D0 85 13 STA $13 ; |_ Zero out 32-bit checksum of 16-bit values.  
E4D2 85 14 STA $14 ; |  
E4D4 85 01 STA $01 ; /
```

```

E4D6 A5 0F LDA $0F ; \
E4D8 8D A6 05 STA $05A6 ; | _ Copy 'addrlo' to our CMD address argument.
E4DB A5 10 LDA $10 ; |
E4DD 8D A5 05 STA $05A5 ; /

E4E0 A9 04 LDA #$04 ; \
E4E2 8D A3 05 STA $05A3 ; | _ Read 16-bit value from CP1610 side (CMD #4)
E4E5 AD A3 05 LDA $05A3 ; |
E4E8 D0 FB BNE $E4E5 ; /

;; Compute 16-bit checksum of 8-bit values from lower bytes.
;; "lower EPROM" checksum
E4EA AD A7 05 LDA $05A7 ; \
E4ED 18 CLC ; |
E4EE 65 0D ADC $0D ; |
E4F0 85 0D STA $0D ; | - Accumulate lower byte checksum.
E4F2 A9 00 LDA #$00 ; |
E4F4 65 0E ADC $0E ; |
E4F6 85 0E STA $0E ; /

;; Compute 16-bit checksum of 8-bit values from upper bytes.
;; "upper EPROM" checksum
E4F8 AD A8 05 LDA $05A8 ; \
E4FB 18 CLC ; |
E4FC 65 0B ADC $0B ; |
E4FE 85 0B STA $0B ; | - Accumulate upper byte checksum.
E500 A9 00 LDA #$00 ; |
E502 65 0C ADC $0C ; |
E504 85 0C STA $0C ; /

;; Compute 32-bit checksum of 16-bit words.
;; "10-bit ROM" checksum, similar to what MTE-201 displays?
E506 AD A7 05 LDA $05A7 ; \
E509 18 CLC ; |
E50A 65 12 ADC $12 ; |
E50C 85 12 STA $12 ; |
E50E AD A8 05 LDA $05A8 ; |
E511 65 13 ADC $13 ; |
E513 85 13 STA $13 ; | - 16 + 32 => 32 accumulation
E515 A9 00 LDA #$00 ; |
E517 65 14 ADC $14 ; |
E519 85 14 STA $14 ; |
E51B A9 00 LDA #$00 ; |
E51D 65 01 ADC $01 ; |
E51F 85 01 STA $01 ; /

E521 A5 06 LDA $06 ; \
E523 C5 0F CMP $0F ; |
E525 D0 48 BNE $E56F ; | _ Have we reached 'addrhi' yet?
E527 A5 07 LDA $07 ; | No: Move to next address.
E529 C5 10 CMP $10 ; |
E52B D0 42 BNE $E56F ; /

```

```

E52D A5 0E LDA $0E ; \
E52F 20 0E EB JSR $EB0E ; |_ Print out lower byte / "low EPROM" checksum
E532 A5 0D LDA $0D ; | to serial.
E534 20 0E EB JSR $EB0E ; /
E537 A9 20 LDA #$20 ; \_ ' ' to serial.
E539 20 AB EA JSR $EAAB ; /
E53C A5 0C LDA $0C ; \
E53E 20 0E EB JSR $EB0E ; |_ Print out upper byte / "high EPROM" checksum
E541 A5 0B LDA $0B ; | to serial.
E543 20 0E EB JSR $EB0E ; /
E546 20 A4 EA JSR $EAA4 ; CR+LF to serial

;; Print the 32-bit checksum out in octal to match how MTE-201
;; would do it (I imagine).
E549 20 85 E5 JSR $E585 ; \
E54C 20 85 E5 JSR $E585 ; | Rotate checksum left by 5.
E54F 20 85 E5 JSR $E585 ; |- $01 $14 $13 $12
E552 20 85 E5 JSR $E585 ; | vutsrqpo:nmlkjihg:fedcba98:76543210
E555 20 85 E5 JSR $E585 ; / qponmlkj:ihgfedcb:a9876543:210vutsr

E558 A9 09 LDA #09 ; \_ 9 octal digits total. (Bottom 27 bits of csum)
E55A 85 02 STA $02 ; /
E55C 20 78 E5 JSR $E578 ; Output octal digit from 3 MSBs of $01
E55F 20 85 E5 JSR $E585 ; \
E562 20 85 E5 JSR $E585 ; |- Rotate left by 3
E565 20 85 E5 JSR $E585 ; /
E568 C6 02 DEC $02 ; \_ Loop over all 9 digits
E56A D0 F0 BNE $E55C ; /
E56C 4C BD E0 JMP $E0BD ; Back to command loop

E56F E6 0F INC $0F ; \
E571 D0 02 BNE $E575 ; |_ Move to next address and continue.
E573 E6 10 INC $10 ; |
E575 4C D6 E4 JMP $E4D6 ; /

E578 A5 01 LDA $01 ; \
E57A 2A ROL A ; |
E57B 2A ROL A ; |- Rotate top 3 bits of A into bottom 3 bits
E57C 2A ROL A ; |
E57D 2A ROL A ; /
E57E 29 07 AND #$07 ; \_ Convert to ASCII '0' .. '7'
E580 09 30 ORA #$30 ; /
E582 4C AB EA JMP $EAAB ; Print, and return back to print-out loop

E585 26 12 ROL $12 ; \
E587 26 13 ROL $13 ; |_ Rotate 32-bit checksum left 1 bit.
E589 26 14 ROL $14 ; |
E58B 26 01 ROL $01 ; /
E58D 60 RTS ; Return

```

\$E58E: Process the 'ZERO' command: Zero out \$5000 - \$6FFF, \$9000 - \$9FFF

```

E58E A9 00 LDA #$00
E590 8D A3 05 STA $05A3 ; \_ Zero out CMD
E593 8D A3 45 STA $45A3 ; /
E596 85 04 STA $04 ; Zero out ?flag?
E598 A2 28 LDX #$28 ; \
E59A A9 00 LDA #$00 ; |
E59C 9D A2 05 STA $05A2,X ; |_ Zero out the CMD argument buffer
E59F 9D A2 45 STA $45A2,X ; | $05A3-$05CA / $45A3-$45CA (6502)
E5A2 CA DEX ; | $85A3-$85CA (CP1610)
E5A3 D0 F7 BNE $E59C ; /

E5A5 A9 50 LDA #$50 ; \_ $05A5:$05A6 = $5000
E5A7 8D A5 05 STA $05A5 ; /
E5AA A9 20 LDA #$20 ; \_ $20 words
E5AC 8D A4 05 STA $05A4 ; /
E5AF 20 C5 E5 JSR $E5C5 ; Issue CMD #01 over 4K of addresses
E5B2 C9 70 CMP #$70 ; \_ Have we zeroed out $5000 - $6FFF?
E5B4 D0 F9 BNE $E5AF ; / No: Keep going.
E5B6 A9 90 LDA #$90 ; \_ Move to $9000
E5B8 8D A5 05 STA $05A5 ; /
E5BB 20 C5 E5 JSR $E5C5 ; \
E5BE C9 A0 CMP #$A0 ; |- Zero $9000 - $9FFF in the same way.
E5C0 D0 F9 BNE $E5BB ; /
E5C2 4C BD E0 JMP $E0BD ; Return to command loop

E5C5 A9 01 LDA #$01 ; \
E5C7 8D A3 05 STA $05A3 ; |_ Issue CMD #1 (copy arg buf)
E5CA AD A3 05 LDA $05A3 ; | to zero out $20 words
E5CD D0 FB BNE $E5CA ; /
E5CF AD A6 05 LDA $05A6 ; \
E5D2 18 CLC ; |
E5D3 D8 CLD ; |- Advance to the next $20 words
E5D4 69 20 ADC #$20 ; |
E5D6 8D A6 05 STA $05A6 ; /
E5D9 D0 EA BNE $E5C5 ; Loop over all 256 words
E5DB AD A5 05 LDA $05A5 ; \
E5DE 69 00 ADC #$00 ; |- Advance upper byte of addr and return
E5E0 8D A5 05 STA $05A5 ; /
E5E3 60 RTS

```

\$E5E4: Validate hex digit, and push into arg at \$06 - \$07; return C=1 on error

```

E5E4 C9 30 CMP #$30 ; \_ A < '0'? Not valid.
E5E6 90 0C BCC $E5F4 ; /
E5E8 C9 3A CMP #$3A ; \_ A < '9'+1? Valid: Push the digit
E5EA 90 0D BCC $E5F9 ; /
E5EC C9 41 CMP #$41 ; \_ A < 'A'? Not valid.
E5EE 90 04 BCC $E5F4 ; /
E5F0 C9 47 CMP #$47 ; \_ A < 'F'+1? Valid: Push the digit

```

```

E5F2 90 02    BCC $E5F6    ; / adjusting for the fact it's alpha.
E5F4 38      SEC          ; C=1 means "bad digit"
E5F5 60      RTS

E5F6 38      SEC          ; \_ Adjust "A..F" down next to digits.
E5F7 E9 37    SBC #$37    ; /

E5F9 48      PHA          ; Save char
E5FA 06 06    ASL $06     ; \
E5FC 26 07    ROL $07     ; |
E5FE 06 06    ASL $06     ; |
E600 26 07    ROL $07     ; |_ Rotate arg buffer left 4 bits
E602 06 06    ASL $06     ; |
E604 26 07    ROL $07     ; |
E606 06 06    ASL $06     ; |
E608 26 07    ROL $07     ; /
E60A 68      PLA          ; Restore char
E60B 29 0F    AND #$0F    ; Keep 8 LSBs
E60D 05 06    ORA $06     ; \_ Merge into bottom of buffer
E60F 85 06    STA $06     ; /
E611 E6 08    INC $08     ; Inc digit count for this argument
E613 18      CLC          ; C=0 means "OK!"
E614 60      RTS

```

\$E615: String: "\024STARTING DOWNLOAD..."

```

E615 .byte $14, $53, $54, $41, $52, $54, $49, $4E, $47, $20 ; .STARTING
E61F .byte $44, $4F, $57, $4E, $4C, $4F, $41, $44, $2E, $2E ; DOWNLOAD..
E629 .byte $2E

```

\$E62A: Process a download record (':')

```

E62A A9 00    LDA #$00
E62C 85 00    STA $00      ;
E62E 85 01    STA $01      ; \_ Zero out checksum
E630 85 11    STA $11      ; /
E632 20 C6 E6 JSR $E6C6    ; Get record address into $0C:$0B (16 bit format)
E635 20 A8 E6 JSR $E6A8    ; Get next declc into $0E:$0D
E638 A6 0C    LDX $0C     ; \
E63A A4 0B    LDY $0B     ; |_ Print address to display
E63C 20 B2 E7 JSR $E7B2    ; /
E63F AD A3 05 LDA $05A3    ; \_ If CMD is busy, print BAD RECORD
E642 D0 45    BNE $E689    ; /
E644 A5 0C    LDA $0C     ; \
E646 8D A5 05 STA $05A5    ; |_ Address for CMD
E649 A5 0B    LDA $0B     ; |
E64B 8D A6 05 STA $05A6    ; /
E64E A5 0E    LDA $0E     ; \
E650 8D A8 05 STA $05A8    ; |_ Data for CMD (big endian order because why?)
E653 A5 0D    LDA $0D     ; |
E655 8D A7 05 STA $05A7    ; /

```

```

E658 A9 05 LDA #$05 ; \_ CMD #5: Write data on CP1610 side
E65A 8D A3 05 STA $05A3 ; /
E65D E6 0B INC $0B ; \
E65F D0 02 BNE $E663 ; |_ Increment address and do next declc
E661 E6 0C INC $0C ; |
E663 4C 35 E6 JMP $E635 ; /

```

\$E666: Process the checksum on a download record (',' portion)

```

E666 A5 11 LDA $11 ; \
E668 48 PHA ; |_ Push the checksum onto the stack
E669 A5 01 LDA $01 ; |
E66B 48 PHA ; /
E66C 20 C6 E6 JSR $E6C6 ; Receive the checksum
E66F 68 PLA ; \
E670 18 CLC ; |
E671 65 0B ADC $0B ; |
E673 85 01 STA $01 ; | Pop the checksum off the stack and add it
E675 68 PLA ; | to what we received. Sum should be $0000.
E676 65 0C ADC $0C ; |
E678 85 11 STA $11 ; |
E67A A5 01 LDA $01 ; | <- E67A alt entry point from alt record fmt.
E67C 05 11 ORA $11 ; /
E67E D0 09 BNE $E689 ; Non-zero: "BAD RECORD..."
E680 20 E5 E7 JSR $E7E5 ; Print "RECORD OK"
E683 A9 06 LDA #$06 ; \_ Send a ^F out the serial port
E685 20 AB EA JSR $EAAB ; /
E688 60 RTS

E689 20 F0 E7 JSR $E7F0 ; Print "BAD RECORD..."
E68C A9 15 LDA #$15 ; \_ Send ^U out serial port. (blocking)
E68E 20 AB EA JSR $EAAB ; / ^U = ASCII NAK.
E691 A9 3E LDA #$3E ; \_ Send '>' out serial port. (blocking)
E693 20 AB EA JSR $EAAB ; /
E696 20 FB E7 JSR $E7FB ; Update error counter onscreen.
E699 20 BC EA JSR $EABC ; get character from serial port
E69C C9 0D CMP #$0D ; \_ If '\r' return.
E69E F0 07 BEQ $E6A7 ; /
E6A0 C9 3A CMP #$3A ; \ If ':' start a new record download,
E6A2 D0 F5 BNE $E699 ; | - otherwise keep draining the serial port
E6A4 4C 2A E6 JMP $E62A ; / until ':' or '\r'.
E6A7 60 RTS

```

\$E6A8: Receive next declc, encoded as 2 chars, into \$0E:\$0D

```

E6A8 20 FB E6 JSR $E6FB ; Get next 5 bits (bits 9:5 of declc)
E6AB 48 PHA ;
E6AC 4A LSR A ; \
E6AD 4A LSR A ; | - Bits [9:8] declc to $0E bits [1:0]
E6AE 4A LSR A ; |
E6AF 85 0E STA $0E ; /

```

```

E6B1 68      PLA      ;
E6B2 0A      ASL A    ; \
E6B3 0A      ASL A    ; |
E6B4 0A      ASL A    ; |
E6B5 0A      ASL A    ; |- Bits [7:5] of decle to $0D bits [7:5]
E6B6 0A      ASL A    ; |
E6B7 29 E0   AND #$E0 ; |
E6B9 85 0D   STA $0D  ; /
E6BB 20 FB E6 JSR $E6FB ; Get next 5 bits (bits 4:0 of decle)
E6BE 05 0D   ORA $0D  ; \_ merge into bits $0D bits [4:0]
E6C0 85 0D   STA $0D  ; /
E6C2 60      RTS

```

\$E6C6: Get record address into \$0C:\$0B, or switch to Intel HEX processing

The entry point to this code is \$E6C6. The JMP instruction at \$E6C3 just extends the branch reach of the conditional branch at \$E6CE.

```

E6C3 4C 32 E7 JMP $E732 ; Extend the reach of conditional branch at E6CE
E6C6 20 FB E6 JSR $E6FB ; Get next 5 bits
E6C9 48      PHA      ; Remember it
E6CA A5 05    LDA $05  ; \ Was last character uppercase?
E6CC C9 40    CMP #$40  ; |- No: This is an Intel HEX record.
E6CE 90 F3    BCC $E6C3 ; / Yes: Handle this as a 16-bit quantity
E6D0 68      PLA      ; Restore it
E6D1 6A      ROR A    ; \
E6D2 6A      ROR A    ; |_ Keep only bit 0, but move it to bit 15 of
E6D3 29 80    AND #$80  ; | the final result.
E6D5 85 0C    STA $0C  ; /
E6D7 20 FB E6 JSR $E6FB ; Get next 5 bits. [14:10] of result
E6DA 0A      ASL A    ; \
E6DB 0A      ASL A    ; |_ Shift it left 2 and merge it into bits
E6DC 05 0C    ORA $0C  ; | [14:10] of the result.
E6DE 85 0C    STA $0C  ; /
E6E0 20 FB E6 JSR $E6FB ; Get next 5 bits. [9:5] of result.
E6E3 48      PHA      ; Save it...
E6E4 4A      LSR A    ; \
E6E5 4A      LSR A    ; |
E6E6 4A      LSR A    ; |- Bits [5:4] of value to bits [9:8] of result.
E6E7 05 0C    ORA $0C  ; |
E6E9 85 0C    STA $0C  ; /
E6EB 68      PLA      ; ...restore it
E6EC 0A      ASL A    ; \
E6ED 0A      ASL A    ; |
E6EE 0A      ASL A    ; |_ Bits [3:0] of value to bits [7:5] of result.
E6EF 0A      ASL A    ; |
E6F0 0A      ASL A    ; |
E6F1 85 0B    STA $0B  ; /
E6F3 20 FB E6 JSR $E6FB ; Get last 5 bits. [4:0] of result.
E6F6 05 0B    ORA $0B  ; \_ merge them in directly.
E6F8 85 0B    STA $0B  ; /
E6FA 60      RTS

```

\$E6FB: Get next nickel of record being downloaded to us; handle ':' and ';' also

```
E6FB 20 BC EA JSR $EABC ; Block waiting for character from serial port
E6FE C9 3A CMP #$3A ; \_ Is it ':'? (Start of new record)
E700 F0 1A BEQ $E71C ; / Yes: Unwind stack and restart ':' command.
E702 C9 3B CMP #$3B ; \_ Is it ';'? (End of current record)
E704 F0 1D BEQ $E723 ; / Yes: Unwind stack and validate the record.
E706 29 7F AND #$7F ; \
E708 C9 20 CMP #$20 ; |- Is it a control char?
E70A 90 1E BCC $E72A ; / Yes: Unwind and report BAD RECORD.
E70C 48 PHA ; Stack it away for a moment.
E70D 18 CLC ; \
E70E 65 01 ADC $01 ; |
E710 85 01 STA $01 ; |_ Update checksum in $11:$01
E712 A9 00 LDA #$00 ; |
E714 65 11 ADC $11 ; |
E716 85 11 STA $11 ; /
E718 68 PLA ; Restore it.
E719 29 1F AND #$1F ; Only keep lower 5 bits.
E71B 60 RTS ;
```

\$E71C: Handle ':' in download record: Unwind stack and restart record

```
E71C 68 PLA ; \
E71D 68 PLA ; |
E71E 68 PLA ; |- Unwind stack and restart ':' command.
E71F 68 PLA ; |
E720 4C 2A E6 JMP $E62A ; /
```

\$E723: Handle ';' in download record: Unwind stack, receive & validate checksum

```
E723 68 PLA ; \
E724 68 PLA ; |
E725 68 PLA ; |- Unwind stack and validate the record.
E726 68 PLA ; |
E727 4C 66 E6 JMP $E666 ; /
```

\$E72A: Unwind stack and report "BAD RECORD..."

```
E72A 68 PLA ; \
E72B 68 PLA ; |
E72C 68 PLA ; |- Unwind stack and report BAD RECORD.
E72D 68 PLA ; |
E72E 4C 89 E6 JMP $E689 ; /
E731 .byte $00 ; Dead code?
```


\$E732: Process Intel HEX record for 6502-side

```
E732 68      PLA          ; \
E733 68      PLA          ; |- Unwind stack. We're shifting gears to IHEX.
E734 68      PLA          ; /
E735 A9 00    LDA #$00    ; \
E737 85 01    STA $01     ; |- Zero out our checksum.
E739 85 11    STA $11     ; /  Structured code?  Chaos is a structure!

E73B A5 05    LDA $05     ; \  Take lower 4 bits of this char, merge it
E73D 29 0F    AND #$0F    ; |- with another 4 bits and re-init our csum.
E73F 20 86 E7 JSR $E786   ; /  (The masking turns out to be unnecessary.)
E742 85 02    STA $02     ; Length of record
E744 20 83 E7 JSR $E783   ; \_ Get next byte to $0C      \
E747 85 0C    STA $0C     ; /                               |_ Our download addr.
E749 20 83 E7 JSR $E783   ; \_ Get next byte to $0B      |
E74C 85 0B    STA $0B     ; /                               /
E74E 20 83 E7 JSR $E783   ; \_ Get next byte to $03.  (IHEX record type, unused.)
E751 85 03    STA $03     ; /

E753 A6 0C    LDX $0C     ; \
E755 A4 0B    LDY $0B     ; |- Print address to CRTM
E757 20 B2 E7 JSR $E7B2   ; /
E75A A5 02    LDA $02     ; \_ Are we at the end of the record?
E75C D0 06    BNE $E764   ; /
E75E 20 83 E7 JSR $E783   ; Get checksum byte
E761 4C 7A E6 JMP $E67A   ; Verify 1-byte checksum

E764 20 83 E7 JSR $E783   ; Get data byte
E767 A0 00    LDY #$00    ; \
E769 91 0B    STA ($0B),Y ; |- Store to address directly (no CMD)
E76B A5 0C    LDA $0C     ; /
E76D 09 40    ORA #$40    ; \
E76F 85 0E    STA $0E     ; |_ Set $0E:$0D to address OR $4000
E771 A5 0B    LDA $0B     ; | This shifts to the bit 9:8 part of memory map
E773 85 0D    STA $0D     ; /
E775 98      TYA          ; \_ Store a zero to bits 9:8 of decle
E776 91 0D    STA ($0D),Y ; /
E778 C6 02    DEC $02     ; Decrement remaining-bytes count.

E77A E6 0B    INC $0B     ; \
E77C D0 D5    BNE $E753   ; |_ Move to next address and loop
E77E E6 0C    INC $0C     ; |
E780 4C 53 E7 JMP $E753   ; /
```

\$E783: Get hexadecimal byte from serial port (blocking)

```
E783 20 99 E7 JSR $E799   ; Get next nibble from serial port.

E786 0A      ASL A        ; \
E787 0A      ASL A        ; |_ Shift 'A' to the upper 4 bits.
```

```

E788 0A      ASL A      ; |
E789 0A      ASL A      ; /
E78A 85 00   STA $00    ; Temporarily stash it
E78C 20 99 E7 JSR $E799  ; Get next nibble from serial port.
E78F 05 00   ORA $00    ; Merge into lower 4 bits.
E791 48      PHA        ; Stash on stack
E792 18      CLC        ; \
E793 65 01   ADC $01    ; |- Add to lower byte of checksum.
E795 85 01   STA $01    ; /
E797 68      PLA        ; Restore from stack
E798 60      RTS

```

\$E799: Get hexadecimal digit from serial port (blocking)

```

E799 20 BC EA JSR $EABC  ; Get character from serial port
E79C C9 20   CMP #$20   ; \_ Non-printing? Pop the stack and abort out.
E79E 90 0D   BCC $E7AD  ; /
E7A0 C9 40   CMP #$40   ; \_ Is it alpha? (A-F)?
E7A2 B0 03   BCS $E7A7  ; / Yes: Shift down to $A - $F range and return
E7A4 29 0F   AND #$0F   ; No: Just mask and return.
E7A6 60      RTS

E7A7 18      CLC        ; \
E7A8 69 09   ADC #$09   ; |- Convert 'A'..'F' to $A - $F.
E7AA 29 0F   AND #$0F   ; /
E7AC 60      RTS        ;

E7AD 68      PLA        ; \
E7AE 68      PLA        ; |
E7AF 68      PLA        ; |- Pop stack and get out.
E7B0 68      PLA        ; |
E7B1 60      RTS        ; /

```

\$E7B2: Print X:Y to the CRTC at \$B820 - \$B823

```

E7B2 98      TYA        ; \_ Save [7:0] (in Y)
E7B3 48      PHA        ; /
E7B4 8A      TXA        ; \_ Save [15:8] (in X)
E7B5 48      PHA        ; /
E7B6 20 D3 E7 JSR $E7D3  ; convert [15:12] to hex
E7B9 8D 20 B8 STA $B820  ; => CRTC @$B820
E7BC 68      PLA        ; restore [15:8]
E7BD 20 D7 E7 JSR $E7D7  ; convert [11:8] to hex
E7C0 8D 21 B8 STA $B821  ; => CRTC @$B821
E7C3 68      PLA        ; restore [7:0]
E7C4 48      PHA        ; save [7:0]
E7C5 20 D3 E7 JSR $E7D3  ; convert [7:4] to hex
E7C8 8D 22 B8 STA $B822  ; => CRTC @$B822
E7CB 68      PLA        ; restore [7:0]
E7CC 20 D7 E7 JSR $E7D7  ; convert [3:0] to hex
E7CF 8D 23 B8 STA $B823  ; => CRTC @$B823

```

```
E7D2 60      RTS
```

\$E7D3: Convert upper nibble of A into a hexadecimal digit (ASCII)

```
E7D3 4A      LSR A      ; \  
E7D4 4A      LSR A      ; |_ Shift upper nibble to lower nibble.  
E7D5 4A      LSR A      ; |  
E7D6 4A      LSR A      ; /  
; fallthru to E7D7 below
```

\$E7D7: Convert lower nibble of A into a hexadecimal digit (ASCII)

```
E7D7 29 0F    AND #$0F    ; Mask away upper bits  
E7D9 C9 0A    CMP #$0A    ; |_ 0 thru 9 or A thru F?  
E7DB B0 03    BCS $E7E0  ; /  
E7DD 09 30    ORA #$30    ; 0 thru 9  
E7DF 60      RTS  
  
E7E0 18      CLC      ; \  
E7E1 D8      CLD      ; | - A thru F.  
E7E2 69 37    ADC #$37    ; /  
E7E4 60      RTS
```

\$E7E5: Print "RECORD OK... " to CRTC at \$B806.

```
E7E5 AD 32 E8  LDA $E832  ; \  
E7E8 A2 E8    LDX #$E8    ; |_ Print "RECORD OK... "  
E7EA A0 33    LDY #$33    ; | to CRTC FB at B806.  
E7EC 20 F1 EA JSR $EAF1    ; /  
E7EF 60      RTS      ;
```

\$E7F0: Print "BAD RECORD... " to CRTC at \$B806

```
E7F0 AD 47 E8  LDA $E847  ; \  
E7F3 A2 E8    LDX #$E8    ; |_ Print "BAD RECORD... "  
E7F5 A0 48    LDY #$48    ; | to CRTC FB at $B806.  
E7F7 20 F1 EA JSR $EAF1    ; /  
E7FA 60      RTS
```

\$E7FB: Update download error count on the screen at CRTC \$B81D - \$BD1E

```
E7FB AD 1D B8  LDA $B81D  ; |_ Is the error counter blank?  
E7FE C9 80    CMP #$80    ; /  
E800 D0 08    BNE $E80A  ; No: Increment it.  
E802 A9 30    LDA #$30    ; \  
E804 8D 1D B8 STA $B81D  ; | - Yes: Print "00".  
E807 8D 1E B8 STA $B81E  ; /  
E80A EE 1E B8 INC $B81E  ; Increment LS digit.
```

```

E80D AD 1E B8 LDA $B81E ; \
E810 C9 3A CMP #$3A ; |- Did it get bigger than '9'? If not, then leave.
E812 90 08 BCC $E81C ; /
E814 A9 30 LDA #$30 ; \
E816 8D 1E B8 STA $B81E ; |- It did: Reset it to 0 and increment MS digit.
E819 EE 1D B8 INC $B81D ; /
E81C 60 RTS

```

\$E81D: Various status strings

It appears the “DONE WITH LOADING” phrase is never actually used.

```

E81D .byte $14, $44, $4F, $4E, $45, $20, $57, $49, $54, $48 ; .DONE WITH
E827 .byte $20, $4C, $4F, $41, $44, $49, $4E, $47, $2E, $20 ; LOADING.
E831 .byte $20, $14, $52, $45, $43, $4F, $52, $44, $20, $4F ; .RECORD 0
E83B .byte $4B, $2E, $2E, $2E, $20, $20, $20, $20, $20, $20 ; K...
E845 .byte $20, $20, $14, $42, $41, $44, $20, $52, $45, $43 ; .BAD REC
E84F .byte $4F, $52, $44, $2E, $2E, $2E, $20, $20, $20, $20 ; ORD...
E859 .byte $20, $20, $20 ;

```

\$E85C: Process ‘U’ command: Upload data from CP1610 to the host

This uses a slightly different data format from the download format or from the Intel HEX format. Each line starts with a ‘:’ followed by the requested data in modified hexadecimal. The line ends with ‘;’ and a checksum in modified hexadecimal. No address information is provided in the record.

Modified hexadecimal just means that the format uses the first 16 entries of the [ASCII to Nickel Conversion Chart](#).

```

E85C A5 06 LDA $06 ; \_ Copy record length to $0C
E85E 85 0C STA $0C ; /

E860 A9 00 LDA #$00 ; \
E862 85 01 STA $01 ; |- Initialize checksum
E864 85 11 STA $11 ; /
E866 A9 00 LDA #$00 ; \
E868 8D A4 45 STA $45A4 ; |- Clear MSBs on CMD args.
E86B 8D A5 45 STA $45A5 ; |
E86E 8D A6 45 STA $45A6 ; /
E871 A5 13 LDA $13 ; \
E873 8D A5 05 STA $05A5 ; |- Copy arg2 to CMD address arg
E876 A5 12 LDA $12 ; | (Big endian because we hate life.)
E878 8D A6 05 STA $05A6 ; /
E87B 38 SEC ; \
E87C A5 0F LDA $0F ; |
E87E E5 12 SBC $12 ; |- More to send?
E880 A5 10 LDA $10 ; | (arg2 == arg1?)
E882 E5 13 SBC $13 ; |
E884 C9 00 CMP #$00 ; /
E886 10 01 BPL $E889 ; Yes: Keep going
E888 60 RTS

```

```

E889 A9 3A LDA #$3A ; \_ Send ':', blocking
E88B 20 AB EA JSR $EAAB ; /
E88E A5 0C LDA $0C ; \_ Copy record length to $02 to use as loop ctr.
E890 85 02 STA $02 ; /
E892 AD A5 05 LDA $05A5 ; \
E895 20 4E E9 JSR $E94E ; |- Addr[15:12] to CRTC @$B820
E898 8D 20 B8 STA $B820 ; /
E89B AD A5 05 LDA $05A5 ; \
E89E 20 52 E9 JSR $E952 ; |- Addr[11:8] to CRTC @$B821
E8A1 8D 21 B8 STA $B821 ; /
E8A4 AD A6 05 LDA $05A6 ; \
E8A7 20 4E E9 JSR $E94E ; |- Addr[7:4] to CRTC @$B822
E8AA 8D 22 B8 STA $B822 ; /
E8AD AD A6 05 LDA $05A6 ; \
E8B0 20 52 E9 JSR $E952 ; |- Addr[3:0] to CRTC @$B823
E8B3 8D 23 B8 STA $B823 ; /
E8B6 A9 04 LDA #$04 ; \
E8B8 8D A3 05 STA $05A3 ; |_ Read from CP1610 via CMD #4
E8BB AD A3 05 LDA $05A3 ; |
E8BE D0 FB BNE $E8BB ; /
E8C0 EE A6 05 INC $05A6 ; \
E8C3 D0 03 BNE $E8C8 ; |- Increment address
E8C5 EE A5 05 INC $05A5 ; /
E8C8 AD A7 05 LDA $05A7 ; \
E8CB 18 CLC ; |
E8CC 65 01 ADC $01 ; |
E8CE 85 01 STA $01 ; |- Add 16-bit data to 16-bit checksum
E8D0 AD A8 05 LDA $05A8 ; |
E8D3 65 11 ADC $11 ; |
E8D5 85 11 STA $11 ; /
E8D7 AD A8 05 LDA $05A8 ; \
E8DA 4A LSR A ; |
E8DB 4A LSR A ; |_ Send data[15:12]
E8DC 4A LSR A ; |
E8DD 4A LSR A ; |
E8DE 20 46 E9 JSR $E946 ; /
E8E1 AD A8 05 LDA $05A8 ; \_ Send data[11:8]
E8E4 20 46 E9 JSR $E946 ; /
E8E7 AD A7 05 LDA $05A7 ; \
E8EA 4A LSR A ; |
E8EB 4A LSR A ; |_ Send data[7:4]
E8EC 4A LSR A ; |
E8ED 4A LSR A ; |
E8EE 20 46 E9 JSR $E946 ; /
E8F1 AD A7 05 LDA $05A7 ; \_ Send data[3:0]
E8F4 20 46 E9 JSR $E946 ; /
E8F7 C6 02 DEC $02 ; Decrement element count.
E8F9 D0 97 BNE $E892 ; More to send? Keep going.

E8FB A9 3B LDA #$3B ; \_ Send ';', blocking
E8FD 20 AB EA JSR $EAAB ; /
E900 A5 11 LDA $11 ; \

```

```

E902 4A      LSR A      ; |
E903 4A      LSR A      ; |
E904 4A      LSR A      ; |
E905 4A      LSR A      ; |
E906 20 46 E9 JSR $E946 ; |
E909 A5 11    LDA $11    ; | Send accumulated checksum as a "hex" code
E90B 20 46 E9 JSR $E946 ; | using alphabet "@ABCDEFGHJKLMNO" rather
E90E A5 01    LDA $01    ; | than "0123456789ABCDEF".
E910 4A      LSR A      ; |
E911 4A      LSR A      ; |
E912 4A      LSR A      ; |
E913 4A      LSR A      ; |
E914 20 46 E9 JSR $E946 ; |
E917 A5 01    LDA $01    ; |
E919 20 46 E9 JSR $E946 ; /
E91C A9 3E    LDA #$3E   ; \ Send '>', blocking
E91E 20 AB EA JSR $EAAB  ; /
E921 20 BC EA JSR $EABC  ; Wait for byte from serial port.
E924 C9 06    CMP #$06   ; \
E926 D0 10    BNE $E938  ; | - ^F means RECORD OK, so print it.
E928 20 E5 E7 JSR $E7E5  ; / Otherwise goto E938.
E92B AD A5 05 LDA $05A5  ;
E92E 85 13    STA $13    ;
E930 AD A6 05 LDA $05A6  ;
E933 85 12    STA $12    ;
E935 4C 60 E8 JMP $E860  ; Keep going...

E938 C9 15    CMP #$15   ; \ Anything other than ^U: Leave.
E93A D0 09    BNE $E945  ; / ^U: Bad record.
E93C 20 FB E7 JSR $E7FB  ; Update on-screen error count
E93F 20 F0 E7 JSR $E7F0  ; Print "BAD RECORD"
E942 4C 60 E8 JMP $E860  ; Keep going...

E945 60      RTS

```

\$E946: Send single hex digit as one of “@ABCDEFGHJKLMNO” out serial

```

E946 29 0F    AND #$0F   ; \ Send single hex digit as one of
E948 09 40    ORA #$40   ; | - "@ABCDEFGHJKLMNO" and return to caller.
E94A 4C AB EA JMP $EAAB  ; /

E94D .byte $00

```

\$E94E: Convert upper nibble of A into a hexadecimal digit (ASCII)

This code is identical to the code at \$E7D3.

```

E94E 4A      LSR A      ; \
E94F 4A      LSR A      ; | Shift upper nibble to lower nibble.
E950 4A      LSR A      ; |
E951 4A      LSR A      ; /

```

```
; fallthru to E952 below
```

\$E952: Convert lower nibble of A into a hexadecimal digit (ASCII)

This code is identical to the code at \$E7D7.

```
E952 29 0F    AND #$0F    ; Mask away upper bits
E954 C9 0A    CMP #$0A    ; \_ 0 thru 9 or A thru F?
E956 B0 03    BCS $E95B  ; /
E958 09 30    ORA #$30    ; 0 thru 9
E95A 60      RTS

E95B 18      CLC          ; \
E95C D8      CLD          ; | - A thru F.
E95D 69 37    ADC #$37    ; /
E95F 60      RTS
```

\$E960: Launch the CP1610 side of the debugger; \$0D holds success/failure

```
E960 48      PHA          ; \
E961 8A      TXA          ; | - Save A, X
E962 48      PHA          ; /
E963 A9 00    LDA #$00    ; \_ Clear CP1610-is-dead flag (?)
E965 85 0D    STA $0D     ; /
E967 8D FA 3F STA $3FFA    ; Clear handshake byte at $BFFA (CP1610)
E96A 20 A3 E9 JSR $E9A3    ; Copy code to CP1610 side.
E96D A5 0D    LDA $0D     ; \_ Did something go wrong? Set CP1610-is-dead
E96F D0 2A    BNE $E99B  ; / and return.
E971 A9 AA    LDA #$AA    ; \_ Initial handshake value is $AA
E973 8D F9 3F STA $3FF9    ; /

E976 A9 00    LDA #$00    ; \
E978 8D 78 04 STA $0478    ; |
E97B A9 00    LDA #$00    ; |   Appears to be request to CP1610 side to jump
E97D 8D 92 03 STA $0392    ; |   to $8800; however, I don't know how this CMD
E980 A9 88    LDA #$88    ; |   method works. $0478 (6502) / $8478 (CP1610)
E982 8D 93 03 STA $0393    ; |   seems to be an interrupt-related handshake.
E985 A9 40    LDA #$40    ; |
E987 8D 78 04 STA $0478    ; /

E98A A9 32    LDA #$32    ; \_ Long delay
E98C 20 ED E9 JSR $E9ED    ; /

E98F A2 FF    LDX #$FF    ; \
E991 AD FA 3F LDA $3FFA    ; |_ Spin waiting for CP1610 side to handshake
E994 C9 55    CMP #$55    ; | with us (CP1610 writes $55 to $BFFA)
E996 F0 07    BEQ $E99F  ; /
E998 CA      DEX          ; \_ Only spin 255 times
E999 D0 F6    BNE $E991  ; /
E99B A9 FF    LDA #$FF    ; \_ Set CP1610-is-dead flag.
E99D 85 0D    STA $0D     ; /
```

```

E99F 68      PLA      ; \
E9A0 AA      TAX      ; |_ Restore X, A and return.
E9A1 68      PLA      ; |
E9A2 60      RTS      ; /

```

\$E9A3: Copy CP1610 code from \$8000-\$8FFF (6502) to \$8800-\$8FFF (CP1610)

This is what makes the contents of U3 visible to the CP1610 side.

```

E9A3 48      PHA
E9A4 98      TYA
E9A5 48      PHA
E9A6 A9 00    LDA #$00    ; \_ Clear CP1610-is-dead flag
E9A8 85 0D    STA $0D     ; /
E9AA 20 CC E9 JSR $E9CC    ; Set up our source/dest pointers

E9AD A0 00    LDY #$00    ; \
E9AF B1 04    LDA ($04),Y ; |
E9B1 91 08    STA ($08),Y ; | Copy 256 decles.
E9B3 B1 06    LDA ($06),Y ; |- ($04) => ($08) copies lower 8 bits
E9B5 91 0A    STA ($0A),Y ; | ($06) => ($0A) copies upper 2 bits
E9B7 C8      INY         ; |
E9B8 D0 F5    BNE $E9AF   ; /

E9BA E6 05    INC $05     ; \
E9BC E6 07    INC $07     ; |_ Increment MSBs of all the addresses
E9BE E6 09    INC $09     ; |
E9C0 E6 0B    INC $0B     ; /
E9C2 A5 05    LDA $05     ; \
E9C4 C9 88    CMP #$88    ; |- Stop when ($04) gets to $8800.
E9C6 90 E7    BCC $E9AF   ; /
E9C8 68      PLA
E9C9 A8      TAY
E9CA 68      PLA
E9CB 60      RTS

```

\$E9CC: Set up addresses for copying CP1610 code from EPROM to RAM

```

E9CC 48      PHA
                ;; Set up the source pointers:
E9CD A9 00    LDA #$00    ; \
E9CF 85 04    STA $04     ; |_ ($04) points to $8000. $8000 - $87FF hold
E9D1 A9 80    LDA #$80    ; | the lower 8 bits of each CP1610 decle.
E9D3 85 05    STA $05     ; /
E9D5 A9 00    LDA #$00    ; \
E9D7 85 06    STA $06     ; |_ ($06) points to $8800. $8800 - $8FFF hold
E9D9 A9 88    LDA #$88    ; | the upper 2 bits of each CP1610 decle.
E9DB 85 07    STA $07     ; /

                ;; Set up the destination pointers

```



```

E9DD A9 00    LDA #00    ; \
E9DF 85 08    STA 08    ; |_ ($08) points to 0800. This is mapped to
E9E1 85 0A    STA 0A    ; | $8800 on the CP1610 side (lower 8 bits)
E9E3 A9 08    LDA 08    ; /
E9E5 85 09    STA 09    ; \
E9E7 A9 48    LDA 48    ; |- ($0A) points to 4800. This is also mapped
E9E9 85 0B    STA 0B    ; / to 8800 on the CP1610 side (upper 2 bits)
E9EB 68        PLA
E9EC 60        RTS

```

\$E9ED: Delay for a long time, proportional to 'A' squared.

```

E9ED 48        PHA
E9EE 48        PHA
E9EF 38        SEC
E9F0 E9 01    SBC 01    SBC #01
E9F2 D0 FB    BNE 9EF2 BNE $E9EF
E9F4 68        PLA
E9F5 38        SEC
E9F6 E9 01    SBC 01    SBC #01
E9F8 D0 F4    BNE 9F8    BNE $E9EE
E9FA 68        PLA
E9FB 38        SEC
E9FC E9 01    SBC 01    SBC #01
E9FE D0 ED    BNE 9FE    BNE $E9ED
EA00 60        RTS

```

Interlude: R6551 Details

The following code sections provide some initialization for the R6551, as well as Device Service Routines for the KC EXEC. (Not that the Black Whale Monitor ended up using them...) Before we get into that code, though, here are some details on the R6551 that will make the code easier to follow.

```

;; ===== ;;
;; M6551 Details. ;;
;; ;;
;; Addr      Write      Read      ;;
;; $B710     Transmit Data Receive Data ;;
;; $B711     Programmed Reset* Status Register ;;
;; $B712     Command Register Command Register ;;
;; $B713     Control Register Control Register ;;
;; ;;
;; *Data is ignored for Programmed Reset ;;
;; ;;
;; Status register layout: ($B711) ;;
;; ;;
;;      7      6      5      4      3      2      1      0      ;;
;;      +-----+-----+-----+-----+-----+-----+-----+ ;;
;;      |  IRQ  | ~DSR | ~DCD | TXDRE | RXDRF | OVERR | FRERR | PAERR | ;;
;;      +-----+-----+-----+-----+-----+-----+-----+ ;;
;;

```

```

;;      IRQ = Interrupt ReQuested                                ;;
;;      ~DSR = Data Set Ready                                    ;;
;;      ~DCD = Device Carrier Detect                            ;;
;;      TXDRE = Transmit Data Register Empty. 1 = Empty, 0 = Not Empty ;;
;;      RXDRF = Receive Data Register Full. 1 = Full, 0 = Not Full ;;
;;      OVERR = Overrun error. 1 = Error, 0 = No error        ;;
;;      FRERR = Framing error. 1 = Error, 0 = No error        ;;
;;      PAERR = Parity error. 1 = Error, 0 = No error          ;;
;;
;; Command Register layout: ($B712)                               ;;
;;
;;      7      6      5      4      3      2      1      0      ;;
;;      +-----+-----+-----+-----+-----+-----+-----+
;;      | Parity Check Ctl. | ECHOE | Transmit Ctl. | RXIRQ | RXEN |
;;      +-----+-----+-----+-----+-----+-----+
;;
;; Parity Check Ctl:                                           ;;
;;      xx0: No parity checking                                 ;;
;;      001: Odd parity on RX and TX                           ;;
;;      011: Even parity on RX and TX                           ;;
;;      101: Mark Parity Bit on TX; checking disabled          ;;
;;      111: Space Parity Bit on TX; checking disabled          ;;
;;
;; ECHOE: RX echo enable. 0 = normal, 1 = RX echo.            ;;
;;
;; Transmit Ctl:                                               ;;
;;      00: TX IRQ disabled, ~RTS high, transmit disabled.    ;;
;;      01: TX IRQ enabled, ~RTS low, transmit enabled.        ;;
;;      10: TX IRQ disabled, ~RTS low, transmit enabled.        ;;
;;      11: TX IRQ disabled, ~RTS low, transmit BREAK.          ;;
;;
;; RXIRQ: Receive interrupt enable. 1 = disabled, 0 = enabled  ;;
;;
;; RXEN: Receiver enable. 0 = disabled, 1 = enabled.           ;;
;;
;; Control Register layout: ($B713)                               ;;
;;
;;      7      6      5      4      3      2      1      0      ;;
;;      +-----+-----+-----+-----+-----+-----+
;;      | STOP | BITS | CLOCK | DIVISOR |
;;      +-----+-----+-----+-----+
;;
;; STOP: Number of stop bits. 0 = 1 stop bit; 1 = 2 stop bits. ;;
;; BITS: Number of data bits. 00 = 8, 01 = 7, 10 = 6, 11 = 5. ;;
;; CLOCK: Clock source. 0 = external clock, 1 = Baud Rate Generator ;;
;; DIVISOR: ... big table, see data sheet.                      ;;
;; =====

```

\$EA01: Device Service Record (DSR) for R6551 serial port.

To fully understand this record's contents, you need to understand the KC EXEC's DSR processing code. This document does not go into those details. Fortunately, the Black Whale monitor does not use the DSR beyond the initial device initialization.

EA01	.byte	\$05		; Device flags: RX/TX enabled, RX/TX IRQ enabled.
EA02	.byte	\$0B		; Channel flags: Bit 3: IRQ Handler routine
				; Bit 1: Output
				; Bit 0: Input
EA03	.byte	\$49, \$05		; RX FIFO data addr: \$0549
EA05	.byte	\$9B, \$05		; TX FIFO data addr: \$059B
EA07	.byte	\$10, \$B7		; Peripheral addr: \$B710
EA09	.byte	\$26, \$EA		; Handler addr: \$EA26
EA0B	.byte	\$00, \$00		
EA0D	.byte	\$52, \$08		; RX, TX FIFO depths
EA0F	.byte	\$02		; DSR #2
EA10	.byte	\$AD, \$11, \$B7		; LDA #\$B711 ; \
EA13	.byte	\$29, \$80		; AND #\$80 ; - Handler stub
EA15	.byte	\$D0		; BNE ...? ; /

\$EA16: Initializer function for R6551 serial port, called during DSR installation

The divisor setting \$1F would establish a bitrate of 19200, if the circuit used a standard 1.8432MHz BRG crystal. However, the R6551 is driven by an 895kHz clock, and so the baud rate is closer to 9600 baud. (~9313 baud, if I did my math right, making it 2.9% slow.)

EA16	4C 19 EA	JMP \$EA19		; This instruction may get copied to RAM first
				; and run from there.
EA19	A0 02	LDY #\$02		
EA1B	BD 40 03	LDA \$0340,X		
EA1E	91 FB	STA (\$FB),Y		; store #\$05(?) to \$B712. RX/TX IRQ enabled
				; RX enabled, RX IRQ enabled
				; TX enabled, TX IRQ enabled
EA20	C8	INY		
EA21	A9 1F	LDA #\$1F		
EA23	91 FB	STA (\$FB),Y		; store #\$1F to \$B713: 19200, 8-N-1
EA25	60	RTS		

\$EA26: DSR interrupt handler for R6551 serial port

EA26	4C 29 EA	JMP \$EA29		; This instruction may get copied to RAM first
				; and run from there.
EA29	78	SEI		; Interrupts disabled
EA2A	A0 01	LDY #\$01		
EA2C	B1 FB	LDA (\$FB),Y		; Read the UART Status Register (\$B711)
EA2E	29 08	AND #\$08		; Is RX Data Reg full? (bit 3)
EA30	D0 23	BNE \$EA55		; Yes: Go receive data
EA32	B1 FB	LDA (\$FB),Y		; Is TX Data Reg empty? (bit 4)

```

EA34 29 10    AND #$10    ;
EA36 D0 08    BNE $EA40    ; Yes: Go see if we have anything more to TX
EA38 A9 40    LDA #$40    ; \
EA3A 85 90    STA $90     ; |- Set C=0 (no error), V=1 (interrupt handled)
EA3C 24 90    BIT $90     ; /
EA3E 58      CLI          ; Interrupts enabled
EA3F 60      RTS

;; Try to transmit data if we have data to transmit
EA40 A9 9B    LDA #$9B    ; \
EA42 85 98    STA $98     ; |_ point to our TX FIFO
EA44 A9 05    LDA #$05    ; |
EA46 85 99    STA $99     ; /
EA48 20 A1 DE JSR $DEA1    ; Get byte from FIFO
EA4B B0 05    BCS $EA52    ; C = 1 means "no data"
EA4D A5 90    LDA $90     ; \_ Send out the byte (TX reg at $B710).
EA4F 8D 10 B7 STA $B710    ; /
EA52 4C 38 EA JMP $EA38    ; Leave DSR interrupt handler

;; Receive an incoming byte if we can.
EA55 B1 FB    LDA ($FB),Y ; Read UART Status Register ($B711)
EA57 29 07    AND #$07    ; Check overrun, framing, parity errors
EA59 D0 11    BNE $EA6C    ; If any are set, go to error handler
EA5B A9 49    LDA #$49    ; \
EA5D 85 98    STA $98     ; |_ Point to our RX FIFO
EA5F A9 05    LDA #$05    ; |
EA61 85 99    STA $99     ; /
EA63 AD 10 B7 LDA $B710    ; Get byte
EA66 20 E0 DE JSR $DEE0    ; Deposit it in our RX FIFO
EA69 4C 38 EA JMP $EA38    ; Leave DSR interrupt handler.

;; Handle overrun/framing/parity errors
EA6C AD 10 B7 LDA $B710    ; Pop byte from RX queue
EA6F 8D 11 B7 STA $B711    ; Trigger a UART reset. (Value written is ignored.)
EA72 A9 05    LDA #$05    ; \_ Re-enable RX/TX, RX/TX IRQ
EA74 8D 12 B7 STA $B712    ; /
EA77 A9 9B    LDA #$9B    ; \
EA79 85 98    STA $98     ; |_ Point to our TX FIFO
EA7B A9 05    LDA #$05    ; |
EA7D 85 99    STA $99     ; /
EA7F A9 15    LDA #$15    ; \
EA81 20 E0 DE JSR $DEE0    ; |_ Enqueue '^U' '>' (0x15, 0x3E).
EA84 A9 3E    LDA #$3E    ; |
EA86 20 E0 DE JSR $DEE0    ; /
EA89 4C 38 EA JMP $EA38    ; Leave DSR interrupt handler.

```

\$EA8C: Set up DSR for R6551 serial port

```

EA8C A9 02    LDA #$02    ; \_ RPC arg1 = 2 (serial port on channel 2)
EA8E 8D 02 03 STA $0302    ; /
EA91 A9 01    LDA #$01    ; \
EA93 8D 03 03 STA $0303    ; |_ RPC arg2: $EA01 (Serial port DSR record)

```

```

EA96 A9 EA    LDA #$EA    ; |
EA98 8D 04 03 STA $0304   ; /
EA9B A9 05    LDA #$05    ; \_ RPC #5
EA9D 8D 01 03 STA $0301   ; /
EAA0 20 07 C3 JSR $C307   ; Make RPC #5 call. (Install/initialize serial port)
EAA3 60      RTS

```

\$EAA4: Send a CR+LF out the serial port (blocking)

```

EAA4 A9 0D    LDA #$0D    ; \_ CR, and block
EAA6 20 AB EA JSR $EAB   ; /
; fall though to $EAA9

```

\$EAA9: Send an LF out the serial port (blocking)

```

EAA9 A9 0A    LDA #$0A    ; LF
; fall through to $EAB

```

\$EAB: Send the byte in 'A' out the serial port (blocking)

```

EAB 48      PHA
EAC 8D 10 B7 STA $B710
EAF A0 01    LDY #$01    ; \
EAB1 B1 FB    LDA ($FB),Y ; |_ Spin until TX data register empty.
EAB3 29 10    AND #$10    ; | ($FB),Y should point to $B711.
EAB5 F0 FA    BEQ $EAB1   ; /
EAB7 20 D2 EA JSR $EAD2
EABA 68      PLA
EABB 60      RTS

```

\$EABC: Block waiting for a byte from the serial port; return byte in A and \$05

```

EABC 20 C9 EA JSR $EAC9   ; \_ Block waiting for a byte to arrive
EABF 90 FB    BCC $EABC   ; /
EAC1 AD 10 B7 LDA $B710   ; Read the byte
EAC4 29 7F    AND #$7F    ; Mask MSB
EAC6 85 05    STA $05    ; Store to $05
EAC8 60      RTS

```

\$EAC9: Test whether data is waiting on serial port; C=1 means "yes"

```

EAC9 A0 01    LDY #$01    ; \_ Read status register.
EACB B1 FB    LDA ($FB),Y ; / ($FB),Y should point to $B711
EACD 6A      ROR A      ; \
EACE 6A      ROR A      ; |_ Deposit RX Data Reg Full into C
EACF 6A      ROR A      ; |
EAD0 6A      ROR A      ; /
EAD1 60      RTS

```

\$EAD2: Receive byte from serial port, non-blocking, with XON/XOFF flow control

```
EAD2 20 C9 EA JSR $EAC9 ; Sample RX Data Reg Full into C
EAD5 90 17 BCC $EAE0 ; No RX data? Leave.
EAD7 20 BC EA JSR $EABC ; Receive the byte.
EADA C9 03 CMP #$03 ; Is it Ctrl-C?
EADC F0 0F BEQ $EAE0 ; Set C=1 and leave
EADE C9 13 CMP #$13 ; Is it Ctrl-S?
EAE0 D0 0C BNE $EAE0 ; No: Leave. C=?
EAE2 20 BC EA JSR $EABC ;
EAE5 C9 11 CMP #$11 ; Is it Ctrl-Q?
EAE7 F0 06 BEQ $EAEF ; Yes: Set C=0 and leave
EAE9 C9 03 CMP #$03 ; Is it Ctrl-C?
EAEB D0 F5 BNE $EAE2 ; No: Spin waiting for Ctrl-Q or Ctrl-C.
EAED 38 SEC ; Set C=1 if Ctrl-C
EAE0 60 RTS ;
EAE7 18 CLC ; Set C=0 if not Ctrl-C
EAE0 60 RTS
```

\$EAF1: Copy string at X:Y, length A, to CRTC frame buffer at \$B806

```
EAF1 8D A4 05 STA $05A4 ; Store length of string to $05A4 (temp var)
EAF4 86 09 STX $09 ; \_ Pointer to string
EAF6 84 08 STY $08 ; /
EAF8 A0 00 LDY #$00 ; \
EAF8 B1 08 LDA ($08),Y ; |
EAF8 99 06 B8 STA $B806,Y ; |_ Copy string to CRTC frame buffer
EAF8 C8 INY ; |
EB00 CE A4 05 DEC $05A4 ; |
EB03 D0 F5 BNE $EAF8 ; /
EB05 60 RTS
```

\$EB06: Print 10-bit hex value via serial port (blocking)

```
EB06 48 PHA ; Save lower 8 bits
EB07 8A TXA
EB08 29 03 AND #$03 ; Keep only 2 LSBs
EB0A 20 17 EB JSR $EB17 ; Print first digit of 10-bit number (bits[9:8])
EB0D 68 PLA ; Restore lower 8 bits
; Fall-thru to $EB0E
```

\$EB0E: Print 8-bit hex value via serial port (blocking)

```
EB0E 48 PHA ; Re-save lower 8 bits
EB0F 6A ROR A ; \
EB10 6A ROR A ; |
EB11 6A ROR A ; |- print next digit (bits[7:4])
EB12 6A ROR A ; |
EB13 20 17 EB JSR $EB17 ; /
```

```
EB16 68      PLA          ; Restore lower 8
; Fall-thru to $EB17
```

\$EB17: Convert nibble to ASCII hex and send to host (blocking)

```
EB17 29 0F      AND #$0F      ; \
EB19 AA          TAX          ; |_ Convert nibble to hex and return via
EB1A BD 20 EB    LDA $EB20,X ; | blocking-send routine.
EB1D 4C AB EA    JMP $EAAB     ; /

EB20 .byte $30, $31, $32, $33, $34, $35, $36, $37, $38, $39 ; 0123456789
EB2A .byte $41, $42, $43, $44, $45, $46                      ; ABCDEF
```

\$EB30: Process commands that start with '\$': \$M, \$B, \$C, \$R, \$D

```
EB30 20 BC EA    JSR $EABC     ; Receive a byte from serial port
EB33 48          PHA          ; Remember it
EB34 20 AB EA    JSR $EAAB     ; Echo it back
EB37 68          PLA          ; Now let's examine it.
EB38 C9 4D      CMP #$4D      ; \ 'M': Go to EBA9
EB3A F0 6D      BEQ $EBA9     ; /
EB3C C9 42      CMP #$42     ; \ 'B': Go to EB4F
EB3E F0 0F      BEQ $EB4F     ; /
EB40 C9 43      CMP #$43     ; \ 'C': Go to EB61
EB42 F0 1D      BEQ $EB61     ; /
EB44 C9 52      CMP #$52     ; \ 'R': Go to EB69
EB46 F0 21      BEQ $EB69     ; /
EB48 C9 44      CMP #$44     ; \ 'D': Go to EB7B
EB4A F0 2F      BEQ $EB7B     ; /
EB4C 4C 50 E1    JMP $E150     ; Error out.
```

\$EB4F: Process '\$B' command: Set breakpoint at address

```
EB4F A5 07      LDA $07       ; \
EB51 8D A5 05   STA $05A5    ; |_ Address to arg buffer
EB54 A5 06      LDA $06       ; |
EB56 8D A6 05   STA $05A6    ; /
EB59 A9 09      LDA #$09     ; \__ CMD #9: Set breakpoint
EB5B 20 00 EC   JSR $EC00    ; /
EB5E 4C BD E0   JMP $E0BD    ; Return to command loop
```

\$EB61: Process '\$C' command: Clear all breakpoints

```
EB61 A9 07      LDA #$07     ; \__ CMD #7: Clear all breakpoints
EB63 20 00 EC   JSR $EC00    ; /
EB66 4C BD E0   JMP $E0BD    ; Return to command loop
```

\$EB69: Process '\$R' command: Remove breakpoint at address

```
EB69 A5 07 LDA $07 ; \  
EB6B 8D A5 05 STA $05A5 ; |_ Address to arg buffer  
EB6E A5 06 LDA $06 ; |  
EB70 8D A6 05 STA $05A6 ; /  
EB73 A9 08 LDA #$08 ; \_ CMD #8: Clear breakpoint at addr  
EB75 20 00 EC JSR $EC00 ; /  
EB78 4C BD E0 JMP $E0BD ; Return to command loop
```

\$EB7B: Process '\$D' command: Display breakpoints

```
EB7B A9 0A LDA #$0A ; \_ CMD #A: Get breakpoint list  
EB7D 20 00 EC JSR $EC00 ; /  
EB80 20 A4 EA JSR $EAA4 ; CR+LF to serial port  
EB83 A9 00 LDA #$00 ; \_ Index pointer / loop counter  
EB85 85 06 STA $06 ; /  
EB87 A4 06 LDY $06 ;  
EB89 B9 A7 05 LDA $05A7,Y ; \  
EB8C 20 0E EB JSR $EB0E ; |  
EB8F A4 06 LDY $06 ; |- Display next breakpoint in hex  
EB91 B9 A8 05 LDA $05A8,Y ; |  
EB94 20 0E EB JSR $EB0E ; /  
EB97 A9 20 LDA #$20 ; \_ Space char  
EB99 20 AB EA JSR $EAAB ; /  
EB9C E6 06 INC $06 ; \_ Increment index by 2 (2 bytes per record)  
EB9E E6 06 INC $06 ; /  
EBA0 A5 06 LDA $06 ;  
EBA2 C9 10 CMP #$10 ; All 8 breakpoints displayed?  
EBA4 D0 E1 BNE $EB87 ; No: keep going  
EBA6 4C BD E0 JMP $E0BD ; Return to command loop
```

\$EBA9: Process '\$M' command: Memory copy

```
EBA9 A5 11 LDA $11 ; \  
EBAB F0 50 BEQ $EBFD ; |  
EBAD A5 14 LDA $14 ; |_ All three args must be provided.  
EBAF F0 4C BEQ $EBFD ; | Otherwise, error out.  
EBB1 A5 08 LDA $08 ; |  
EBB3 F0 48 BEQ $EBFD ; /  
EBB5 A0 00 LDY #$00 ; Dead code?  
EBB7 A5 0F LDA $0F ; \  
EBB9 8D A6 05 STA $05A6 ; |_ src to arg buffer  
EBBC A5 10 LDA $10 ; |  
EBBE 8D A5 05 STA $05A5 ; /  
EBC1 A9 04 LDA #$04 ; \  
EBC3 8D A3 05 STA $05A3 ; |_ Read address on CP1610 side (CMD #4)  
EBC6 AD A3 05 LDA $05A3 ; |  
EBC9 D0 FB BNE $EBC6 ; /  
EBCB A5 12 LDA $12 ; \  
;
```



```

EBCD 8D A6 05 STA $05A6 ; |_ dst to arg buffer
EBD0 A5 13 LDA $13 ; |
EBD2 8D A5 05 STA $05A5 ; /
EBD5 A9 05 LDA #$05 ; \
EBD7 8D A3 05 STA $05A3 ; |_ Write address on CP1610 side (CMD #5)
EBDA AD A3 05 LDA $05A3 ; | This copies data from src to dst.
EBDD D0 FB BNE $EBDA ; /
EBDF A5 0F LDA $0F ; \
EBE1 C5 06 CMP $06 ; |
EBE3 D0 09 BNE $EBEE ; | - src == end? No: keep going
EBE5 A5 10 LDA $10 ; |
EBE7 C5 07 CMP $07 ; |
EBE9 D0 03 BNE $EBEE ; /
EBEB 4C BD E0 JMP $E0BD ; Return to command loop

EBEE E6 0F INC $0F ; \
EBF0 D0 02 BNE $EBF4 ; | - Increment src
EBF2 E6 10 INC $10 ; /
EBF4 E6 12 INC $12 ; \
EBF6 D0 BF BNE $EBB7 ; | - Increment dst
EBF8 E6 13 INC $13 ; /
EBFA 4C B7 EB JMP $EBB7 ; Keep copying

EBFD 4C 50 E1 JMP $E150 ; Error out.

```

\$EC00: Send CMD in 'A' to CP1610 side; spin until CP1610 indicates it's done

```

EC00 8D A3 05 STA $05A3 ; Write CMD number to shared CMD word
EC03 AD A3 05 LDA $05A3 ; \__ Spin until it reads as zero
EC06 D0 FB BNE $EC03 ; /
EC08 60 RTS

```

\$EC09 - \$EFFF: Empty ROM

The address space \$EC09 - \$EFFF is filled with \$FF.

\$F000 - \$FFFF: Intellivision BASIC ROM

The ROM space from \$F000 - \$FFFF holds the upper half of the Intellivision BASIC ROM. The Black Whale monitor code ignores this ROM. The only portion of this ROM that sees any use are the vectors at \$FFFA - \$FFFF.

```

FFFA .byte $00, $C0
FFFC .byte $00, $C0
FFFE .byte $09, $C0

```

Black Whale Monitor, CP1610 Side

Important CP1610-side variables

Address(es)		Purpose
\$85A3		Command word to CP1610-side. (\$05A3/\$45A3 on 6502 side.)
\$85A4		8-bit argument exchange with 6502-side. (Loaded into R3 on CP1610.)
\$85A5	\$85A6	16-bit argument exchange with 6502-side. (Loaded into R1 on CP1610.)
\$85A7	\$85CA	Variable length argument buffer shared with 6502-side. (R4=#\$85A7 on CP1610.)
\$85CF	\$85E7	Breakpoint buffer
\$85CF		Number of active breakpoints
\$85D0+N*3		Details for breakpoint #N, up to 8 breakpoints. Address + saved opcode
\$85E8	\$85F9	Saved CPU state at breakpoint
\$85E8-\$85E9		Saved processor flags
\$85EA-\$85EB		Saved value of R0
\$85EC-\$85ED		Saved value of R1
\$85EE-\$85EF		Saved value of R2
\$85F0-\$85F1		Saved value of R3
\$85F2-\$85F3		Saved value of R4
\$85F4-\$85F5		Saved value of R5
\$85F6-\$85F8		Saved value of R6 (SP)
\$85F8-\$85F9		Saved value of R7 (PC)
\$85FA	\$8602	Single-step / step-to state:
\$85FA		Single-stepped opcode
\$85FB-\$85FC		Single-step address
\$85FD		Opcode of next instruction
\$85FE-\$85FF		Address of next instruction
\$8600		Opcode of branch target (used if stepping over conditional branch)
\$8601-\$8602		Address of branch target
\$8603		If non-zero, halt after single step; else, report state and continue.
\$8604	\$8606	Self-modifying code for single-stepping instruction.
\$BFF9	\$BFFA	Handshake with 6502 side during initial bootup.

Subroutine / branch target list

For commands issued from 6502-side:

- R3 is initialized to the 8-bit argument passed in \$85A4
- R1 is initialized to the 16-bit argument passed in \$85A5 - \$85A6
- R4 is initialized to the value #\$85A7

The command descriptions below for CMD #\$01 - CMD #\$0F assume that initialization.

Address	Description
\$8800	Initialize CP1610 side of debugger
\$887B	Main command dispatch loop
\$88C5	CMD #\$01: Copy R3 words from argument buffer @R4 to @R1
\$88CF	CMD #\$02: Display string from argument buffer @R4; Length in R3
\$88E9	CMD #\$03: Read a decl @R1, and write to argument buffer @R4
\$88ED	CMD #\$04: Read a 16-bit value @R1 and write to argument buffer @R4, little endian
\$88F4	CMD #\$05: Write a 16-bit value from argument buffer @R4 to location @R1
\$88FA	CMD #\$06: Resume execution at address in R1
\$8944	CMD #\$07: Disable all breakpoints
\$8951	CMD #\$08: Clear breakpoint at address in R1 ¹⁰
\$8970	CMD #\$09: Set breakpoint at address in R1
\$89A0	CMD #\$0A: Copy breakpoint address list to 6502 side
\$89BD	CMD #\$0C: Copy saved CP1610 registers to 6502 side
\$89DA	CMD #\$0D: Set saved register value; Reg # in R3, value in R1
\$89F4	CMD #\$0E: Single step instruction, or run to address in R1 if R1 ≠ 0
\$8A26	CMD #\$0F: Resume execution at currently saved PC
\$8A2E	Get arguments for command
\$8A3F	Clear the command byte and return to command dispatch loop
\$8A47	Alias for \$8A3F
\$8A4A	Copy original code to breakpoint buffer and install breakpoints
\$8A65	Set up SIN2 on the other side of the next instruction (single step)
\$8C04	Restore machine state and prepare to jump into the program

¹⁰ This command's implementation appears buggy. See description alongside the disassembly.

\$8C2B	Interrupt service routine: Detect breakpoints
\$8CC0	Re-enter command dispatch loop, overriding any command request with NOP
\$8CD1	Resume execution after a single step
\$8CF1	Capture some state to prepare to single-step over a breakpoint
\$8D04	Branch to default EXEC ISR at \$1126
\$8D07	Replace breakpoints w/ original opcodes/data
\$8D1F	Restore opcodes after a single step

\$8800: Initialize CP1610 side of debugger

```

L_8800: DIS                ; 8800
        MVII    #$0001, R0   ; 8801 \
        SDBD                ; 8803 |_ Init $8478 to 1.
        MVII    #$8478, R1   ; 8804 | (Addr just before ISR vec.)
        MVO@    R0,    R1    ; 8807 /
        MVII    #$0055, R0   ; 8808 \
        SDBD                ; 880A |_ $55 to $BFFA. Handshake
        MVII    #$BFFA, R1   ; 880B | to say we're alive now.
        MVO@    R0,    R1    ; 880E /

        EIS                ; 880F

        JSR     R5,    G_7166 ; 8810 Clears $847E and R0
        JSR     R5,    G_7535 ; 8813 Flush I/O to 6502-side
        JSR     R5,    G_73CD ; 8816 Clear the screen
        JSR     R5,    G_73D0 ; 8819 Display string
        DECLE  $0280, $0000 ; 881C => Clear "display paused" flag

        SDBD                ; 881E \
        MVII    #$0102, R4   ; 881F |_ Quench the EXEC ISR.
        MVII    #$0081, R0   ; 8822 | Set it to return immediately.
        MVO@    R0,    R4    ; 8824 /

        JSR     R5,    G_73D0 ; 8825 Print string
        DECLE  $0103         ; Set cursor column 3
        DECLE  $0143         ; Set cursor row 3
        STRING "WELCOME TO 1610 DEVELOPMENT SYSTEM", $0D, $0A
        STRING "  VERSION 5", 0

        SDBD                ; 885B \
        MVII    #$8478, R1   ; 885C |_ Clear $8478.
        CLRR    R0           ; 885F | Is this related to interrupts?
        MVO@    R0,    R1    ; 8860 /
        DIS                ; 8861 \
        SDBD                ; 8862 |
        MVII    #$002B, R0   ; 8863 |
        MVO     R0,    .ISRVEC.0 ; 8866 | Painfully set up our ISR
        SDBD                ; 8868 |_ vector to point to $8C2B.
        MVII    #$008C, R0   ; 8869 |
        MVO     R0,    .ISRVEC.1 ; 886C |
        SDBD                ; 886E |
        MVII    #$02F1, R6   ; 886F |
        EIS                ; 8872 /
        JSR     R4,    L_8D07 ; 8873 Remove all breakpoints
        JSR     R4,    L_8D1F ; 8876 Remove all single-step traps

        B       CMD_DONE    ; 8879 Zero out command byte and go
        ;                  ; to command dispatch loop below.
        ;                  ; CMD_DONE = $8A3F

```

\$887B: Main command dispatch loop

```
CMD_DISP:
L_887B: SDBD                ; 887B
        MVII    #$85A3, R1    ; 887C
        MVI@   R1,    R0      ; 887F
        CMPI   #$0000, R0     ; 8880
        BEQ    CMD_DISP      ; 8882

        JSR    R5,    GET_ARGS ; 8884

        CMPI   #$0001, R0     ; 8887
        BEQ    CMD_1         ; 8889

        CMPI   #$0002, R0     ; 888B
        BEQ    CMD_2         ; 888D

        CMPI   #$0003, R0     ; 888F
        BEQ    CMD_3         ; 8891

        CMPI   #$0004, R0     ; 8893
        BEQ    CMD_4         ; 8895

        CMPI   #$0005, R0     ; 8897
        BEQ    CMD_5         ; 8899

        CMPI   #$0006, R0     ; 889B
        BEQ    CMD_6         ; 889D

        CMPI   #$0007, R0     ; 889F
        BEQ    CMD_7         ; 88A1

        CMPI   #$0008, R0     ; 88A3
        BEQ    CMD_8         ; 88A5

        CMPI   #$0009, R0     ; 88A7
        BEQ    CMD_9         ; 88A9

        CMPI   #$000A, R0     ; 88AB
        BEQ    CMD_A         ; 88AD

        CMPI   #$000B, R0     ; 88AF  \_ 6502 watches for this command
        BEQ    CMD_DISP      ; 88B1  /  when trying to halt CP1610.

        CMPI   #$000C, R0     ; 88B3
        BEQ    CMD_C         ; 88B5

        CMPI   #$000D, R0     ; 88B7
        BEQ    CMD_D         ; 88B9

        CMPI   #$000E, R0     ; 88BB
        BEQ    CMD_E         ; 88BD

        CMPI   #$000F, R0     ; 88BF
        BEQ    CMD_F         ; 88C1
        B      CMD_DONE      ; 88C3  Out of range: clear and loop. ($8A3F)
```

\$88C5: CMD #01: Copy R3 words from argument buffer @R4 to @R1

```
CMD_1:
L_88C5: MOVR    R1,    R5        ; 88C5
L_88C6: MVI@   R4,    R1        ; 88C6
        MVO@   R1,    R5        ; 88C7
        DECR  R3         ; 88C8
        CMPI  #$0000, R3      ; 88C9
        BNEQ  L_88C6      ; 88CB
        B     CMD_DONE     ; 88CD Done w/ command. (CMD_DONE=$8A3F)
```

\$88CF: CMD #02: Display string from argument buffer @R4; Length in R3

```
CMD_2:
L_88CF: MOVR    R1,    R5        ; 88CF
L_88D0: MVI@   R5,    R0        ; 88D0
        ANDI  #$007F, R0      ; 88D1 Mask hi-bits on character.

        PSHR  R5         ; 88D3 \
        JSR  R5,    G_73DC     ; 88D4 |- Display character
        PULR  R5         ; 88D7 /

        DECR  R3         ; 88D8
        CMPI  #$0000, R3      ; 88D9
        BNEQ  L_88D0      ; 88DB

        JSR  R5,    G_73D8     ; 88DD CR+LF
        JSR  R5,    G_73D0     ; 88E0 Print "  "
        DECL  $20, $20, $20, 0 ; 88E3 " ",0

        B     CMD_DONE     ; 88E7 Done w/ command. (CMD_DONE=$8A3F)
```

\$88E9: CMD #03: Read a decle @R1, and write to argument buffer @R4

```
CMD_3:
L_88E9: MVI@   R1,    R0        ; 88E9
        MVO@   R0,    R4        ; 88EA
        B     CMD_DONE     ; 88EB Done w/ command. (CMD_DONE=$8A3F)
```

\$88ED: CMD #04: Read a 16-bit value @R1 and write to argument buffer @R4

```
CMD_4:
L_88ED: MOVR    R1,    R5        ; 88ED
        MVI@   R5,    R0        ; 88EE
        MVO@   R0,    R4        ; 88EF
        SWAP  R0,    1         ; 88F0
        MVO@   R0,    R4        ; 88F1
        B     CMD_DONE     ; 88F2 Done w/ command. (CMD_DONE=$8A3F)
```

\$88F4: CMD #05: Write a 16-bit value from argument buffer @R4 to loc @R1

```

CMD_5:
L_88F4: MOVR    R1,    R5            ; 88F4
        SDBD                ; 88F5
        MVI@   R4,    R0            ; 88F6
        MVO@   R0,    R5            ; 88F7
        B      CMD_DONE           ; 88F8 Done w/ command. (CMD_DONE=$8A3F)

```

\$88FA: CMD #06: Resume execution at address in R1

```

CMD_6:
L_88FA: SDBD                ; 88FA \
        MVII   #$85F8, R4      ; 88FB |
        MOVR  R1,    R2        ; 88FE | Write R1 to $85F8 - $85F9
        ANDI  #$00FF, R1      ; 88FF | being careful to strip MSBs
        MVO@  R1,    R4        ; 8901 | from each half. This sets our
        SWAP  R2,    1         ; 8902 | saved R7 to the provided arg.
        ANDI  #$00FF, R2      ; 8903 |
        MVO@  R2,    R4        ; 8905 /
        SDBD                ; 8906 \
        MVII   #$85F6, R4      ; 8907 |
        MOVR  R6,    R1        ; 890A | Write R6 to $85F6 - $85F7
        MOVR  R1,    R2        ; 890B | being careful to strip MSBs
        ANDI  #$00FF, R1      ; 890C | from each half. This shifts
        MVO@  R1,    R4        ; 890E | the saved R6 to our current R6.
        SWAP  R2,    1         ; 890F |
        ANDI  #$00FF, R2      ; 8910 |
        MVO@  R2,    R4        ; 8912 /

        JSR   R4,    L_8A4A    ; 8913 Install breakpoints

        SDBD                ; 8916 \_ Address we're jumping to
        MVII   #$85F8, R4      ; 8917 /
        SDBD                ; 891A \_ Address of single-step point
        MVII   #$85FB, R5      ; 891B /
        SDBD                ; 891E \_ R2 = single-step point
        MVI@   R5,    R2        ; 891F /
        SDBD                ; 8920 \_ R1 = new target
        MVI@   R4,    R1        ; 8921 /
        CMPR  R2,    R1        ; 8922 Are they equal?
        BNEQ  L_8930          ; 8923 No: then prepare to run
        ;                               ; Yes: then prepare to step
        SDBD                ; 8925 \
        MVII   #$85FA, R5      ; 8926 |_ Restore single-step opcode
        MVI@   R5,    R0        ; 8929 |
        MVO@   R0,    R1        ; 892A /

        INCR  R1                ; 892B \
        MVI@  R1,    R3        ; 892C |- Set up SIN,2 on other side of
        JSR   R4,    L_8A65    ; 892D / the single-step.

```



```

L_8930: SDBD                ; 8930 \ Is it the address of
        CMPI   #$1041, R1   ; 8931 | - post-$7xxx init code in the
        BNEQ   L_893F       ; 8934 / EXEC?

        JSR    R5,    $1A83   ; 8936 Clear sound

        SDBD                ; 8939 \
        MVII   #$5014, R0    ; 893A | - Set up cartridge header
        MVO    R0,    G_02F0 ; 893D /

L_893F: JSR    R4,    L_8C04   ; 893F Restore processor state, except R4

        PULR   R4           ; 8942 Restore R4
        PULR   R7           ; 8943 Leap to game.

```

\$8944: CMD #07: Disable all breakpoints

```

CMD_7:
L_8944: CLRR    R0           ; 8944
        MVII   #$0019, R1   ; 8945 \
        SDBD                ; 8947 | - Clear breakpoint buffer from
        MVII   #$85CF, R5   ; 8948 / $85CF - $85E7
L_894B: MVO@   R0,    R5     ; 894B
        DECR   R1           ; 894C
        BNEQ   L_894B       ; 894D
        B      CMD_DONE     ; 894F (CMD_DONE=$8A3F)

```

\$8951: CMD #08: Clear breakpoint at address in R1

It appears this particular routine is buggy. The breakpoint table is allowed to be *sparse*. There are 8 entries. Inactive entries set their breakpoint address to 0.

Nonetheless, the breakpoint table maintains a count of active breakpoints. The breakpoint removal routine below uses that count of active breakpoints as the total trip count for its removal loop.

As a result, if you remove a breakpoint and make the breakpoint table sparse, you will lose access to breakpoints at the end of the table for subsequent calls to this command. It appears that only the remove-breakpoints command is affected by this bug.

```

CMD_8:
L_8951: SDBD                ; 8951 \
        MVII   #$85CF, R5   ; 8952 |_ Are there any active
        MOVR   R5,    R3     ; 8955 | breakpoints?
        MVI@   R5,    R0     ; 8956 /
        CMPI   #$0000, R0    ; 8957 \_ No: CMD_DONE via CMD_DONE_X
        BEQ    CMD_DONE_X    ; 8959 / (CMD_DONE_X=$8A47)

```

```

L_895B: SDBD                ; 895B \
        MVI@   R5,      R2    ; 895C |_ Look for record matching @R1
        CMPR   R1,      R2    ; 895D |
        BEQ    L_8966        ; 895E /

;; There is a bug here.  If R2 == 0, we should skip the entry w/out
;; decrementing R0.  The table is allowed to be sparse, but this
;; decrement assumes it is dense.
        DECR   R0                ; 8960 \_ Number of iterations is the same
        BEQ    CMD_DONE_X        ; 8961 / as the number of active records.

        INCR   R5                ; 8963 \_ Skip saved opcode in
        B      L_895B            ; 8964 / breakpoint record.

L_8966: SUBI    #$0002, R5      ; 8966 \
        CLRR   R2                ; 8968 |_ Clear the address assoc w/
        MVO@   R2,      R5      ; 8969 | this record, inactivating it.
        MVO@   R2,      R5      ; 896A /
        MVI@   R3,      R0      ; 896B \
        DECR   R0                ; 896C |_ Decrement active record count.
        MVO@   R0,      R3      ; 896D /
        B      CMD_DONE        ; 896E Leave. (CMD_DONE=$8A3F)

```

\$8970: CMD #\$09: Set breakpoint at address in R1

```

CMD_9:
L_8970: SDBD                ; 8970 \
        MVII   #$85CF, R5      ; 8971 |
        MOVR   R5,      R4      ; 8974 |- Are there already 8 active
        MVI@   R5,      R0      ; 8975 | entries in the buffer?
        CMPI   #$0008, R0      ; 8976 | If so, leave.
        BEQ    CMD_DONE_X        ; 8978 /

        MVII   #$0008, R2      ; 897A Iterate over all 8 slots
L_897C: SDBD                ; 897C \
        MVI@   R5,      R3      ; 897D |- Does this slot match address
        CMPR   R1,      R3      ; 897E / in R1?
        BEQ    CMD_DONE_X        ; 897F If so, leave.

        INCR   R5                ; 8981 Skip data assoc. with address.
        DECR   R2                ; 8982 \_ loop over remaining entries.
        BNEQ   L_897C            ; 8983 /

        SDBD                ; 8985
L_8989: MVII   #$85D0, R5      ; 8986
        SDBD                ; 8989 \
        MVI@   R5,      R3      ; 898A |_ Is this an empty slot? If so,
        CMPI   #$0000, R3      ; 898B | break out.
        BEQ    L_8992            ; 898D /

        INCR   R5                ; 898F \_ Loop over all the slots.
        B      L_8989            ; 8990 /

```

```

L_8992: SUBI    #$0002, R5          ; 8992  \
        MOVR   R1,    R2          ; 8994  |
        ANDI   #$00FF, R1        ; 8995  |
        MVO@   R1,    R5          ; 8997  |- Store our address from R1
        SWAP   R2,    1           ; 8998  | in the slot.
        ANDI   #$00FF, R2        ; 8999  |
        MVO@   R2,    R5          ; 899B  /
        INCR   R0           ; 899C  \_ Store the updated number of
        MVO@   R0,    R4          ; 899D  / active slots.
        B      CMD_DONE          ; 899E  Leave. (CMD_DONE=$8A3F)

```

\$89A0: CMD #\$0A: Copy breakpoint address list to 6502 side

```

CMD_A:
L_89A0: MVII   #$0008, R3          ; 89A0  Look at all 8 records
        SDBD           ; 89A2  \_ Breakpoint buffer
        MVII   #$85D0, R5        ; 89A3  /
        SDBD           ; 89A6  \_ variable argument/return buffer
        MVII   #$85A7, R4        ; 89A7  /
L_89AA: SDBD           ; 89AA  \_ Read address from bkpt record
        MVI@   R5,    R0          ; 89AB  / buffer.
        MOVR   R0,    R1          ; 89AC  \
        SWAP   R1,    1           ; 89AD  |
        ANDI   #$00FF, R1        ; 89AE  |_ Store it MSB first. ??
        MVO@   R1,    R4          ; 89B0  |
        ANDI   #$00FF, R0        ; 89B1  |
        MVO@   R0,    R4          ; 89B3  /
        INCR   R5           ; 89B4  Skip data value
        DECR   R3           ; 89B5  \
        CMPI   #$0000, R3        ; 89B6  |- Loop over all slots
        BNEQ   L_89AA          ; 89B8  /
        B      CMD_DONE          ; 89BA  Leave. (CMD_DONE=$8A3F)

        NOP           ; 89BC  An orphan?

```

\$89BD: CMD #\$0C: Copy saved CP1610 registers to 6502 side

```

CMD_C:
L_89BD: SDBD           ; 89BD
        MVII   #$85A7, R4        ; 89BE
        SDBD           ; 89C1
        MVII   #$85E8, R5        ; 89C2
        MVII   #$0009, R3        ; 89C5

L_89C7: SDBD           ; 89C7  \
        MVI@   R5,    R0          ; 89C8  |
        MOVR   R0,    R1          ; 89C9  | Carefully copy a 16-bit value
        SWAP   R1,    1           ; 89CA  |_ from the saved processor state
        ANDI   #$00FF, R1        ; 89CB  | to the arg reply buffer.
        MVO@   R1,    R4          ; 89CD  | (Big endian byte order.)

```

```

ANDI    #$00FF, R0      ; 89CE  |
MVO@    R0,    R4      ; 89D0  /
DECR    R3              ; 89D1
CMPI    #$0000, R3     ; 89D2
BNEQ    L_89C7         ; 89D4

MVI@    R5,    R0      ; 89D6
MVO@    R0,    R4      ; 89D7
B       CMD_DONE      ; 89D8  (CMD_DONE=$8A3F)

```

\$89DA: CMD #\$0D: Set saved register value; Reg # in R3, value in R1

```

CMD_D:
L_89DA: CMPI    #$0008, R3      ; 89DA  \_ R0 - R7: Set saved proc register
        BNEQ    L_89E4         ; 89DC  / #8: Set saved processor flags

        SDBD                ; 89DE  \
MVII    #$85E8, R4           ; 89DF  |- Point to saved proc flags
B       L_89EA              ; 89E2  /

L_89E4: SDBD                ; 89E4  \
MVII    #$85EA, R4           ; 89E5  |_ point to saved processor reg
SLL    R3,    1              ; 89E8  |
ADDR   R3,    R4            ; 89E9  /

L_89EA: MOVR    R1,    R0      ; 89EA  \
ANDI    #$00FF, R0          ; 89EB  |
MVO@    R0,    R4          ; 89ED  |_ Save new register value.
SWAP    R1,    1           ; 89EE  |
ANDI    #$00FF, R1          ; 89EF  |
MVO@    R1,    R4          ; 89F1  /
B       CMD_DONE          ; 89F2  (CMD_DONE=$8A3F)

```

\$89F4: CMD #\$0E: Single step instruction, or run to address in R1 if R1 ≠ 0

```

CMD_E:
L_89F4: SDBD                ; 89F4  \
MVII    #$8603, R5          ; 89F5  |_ Set halt after single step flag.
MVII    #$0001, R0          ; 89F8  |
MVO@    R0,    R5          ; 89FA  /
SDBD                ; 89FB  \_ Address of saved PC.
MVII    #$85F8, R4          ; 89FC  /
CMPI    #$0000, R1          ; 89FF  \_ Did user supply a run-until
BNEQ    L_8A07             ; 8A01  / address? Yes: Goto L_8A07

        SDBD                ; 8A03  \
MVI@    R4,    R1          ; 8A04  |_ Get saved PC into R1.
DECR    R4              ; 8A05  |
DECR    R4              ; 8A06  /
L_8A07: MVI@    R1,    R0      ; 8A07  Get target opcode
INCR    R1              ; 8A08  \

```

```

MVI@ R1, R3 ; 8A09 |- Get next word into R3.
DECR R1 ; 8A0A / Is this dead code?
MOVR R1, R2 ; 8A0B \
ANDI #$00FF, R1 ; 8A0C |
MVO@ R1, R4 ; 8A0E |- Save address of instruction
SWAP R2, 1 ; 8A0F |
ANDI #$00FF, R2 ; 8A10 |
MVO@ R2, R4 ; 8A12 /
SUBI #$0004, R4 ; 8A13 Point back to saved R6.
MOVR R6, R1 ; 8A15 Copy our R6 to R1
MOVR R1, R2 ; 8A16 \
ANDI #$00FF, R1 ; 8A17 |
MVO@ R1, R4 ; 8A19 |- Make our R6 the saved R6.
SWAP R2, 1 ; 8A1A |
ANDI #$00FF, R2 ; 8A1B |
MVO@ R2, R4 ; 8A1D /

JSR R4, L_8A65 ; 8A1E Set up other side of single-step
JSR R4, L_8C04 ; 8A21 Prepare to resume program

PULR R4 ; 8A24 Restore R4
PULR R7 ; 8A25 Flying leap back to program

```

\$8A26: CMD #\$0F: Resume execution at currently saved PC

```

CMD_F:
L_8A26: SDBD ; 8A26 \
MVII #$85F8, R4 ; 8A27 |- Get saved PC into R1
SDBD ; 8A2A |
MVI@ R4, R1 ; 8A2B /
B CMD_6 ; 8A2C Continue w/ CMD_6 (CMD_6=$88FA)

```

\$8A2E: Get arguments for command

```

GET_ARGS:
L_8A2E: SDBD ; 8A2E
MVII #$85A4, R1 ; 8A2F
MVI@ R1, R3 ; 8A32 R3 = @$85A4
SDBD ; 8A33
MVII #$85A5, R4 ; 8A34 \
SDBD ; 8A37 |- R1 = (@$85A5 << 8) | $85A6
MVI@ R4, R1 ; 8A38 | Opposite of SDBD order.
SWAP R1, 1 ; 8A39 /
SDBD ; 8A3A

MVII #$85A7, R4 ; 8A3B R4 = addr of bidir arg buffer.
MOVR R5, R7 ; 8A3E

```

\$8A3F: Clear the command byte and return to the command dispatch loop

```
CMD_DONE:
L_8A3F: CLRR    R2                ; 8A3F  \
          SDBD                ; 8A40  |_ Clear command byte
          MVII    #$85A3, R1    ; 8A41  |
          MVO@   R2,    R1     ; 8A44  /
          B      CMD_DISP      ; 8A45  Branch to dispatch loop (CMD_DISP=$887B)
```

\$8A47: Alias for \$8A3F

```
CMD_DONE_X:
L_8A47: NOP                ; 8A47  \_ A real round about way to
          B      CMD_DONE    ; 8A48  / branching to CMD_DONE.
```

\$8A4A: Copy original code to breakpoint buffer and install breakpoints

```
L_8A4A: SDBD                ; 8A4A
          MVII    #$85CF, R5  ; 8A4B  Breakpoint buffer
          MVII    #$0036, R2  ; 8A4E  SIN opcode
          MVI@   R5,    R0    ; 8A50  Number of active breakpoints
L_8A51: CMPI    #$0000, R0    ; 8A51  \_ No more active records? Leave.
          BEQ    L_8A64      ; 8A53  /

L_8A55: SDBD                ; 8A55  \_ Get address
          MVI@   R5,    R1    ; 8A56  /
          CMPI    #$0000, R1  ; 8A57  NULL? Skip it
          BNEQ   L_8A5E      ; 8A59

          INCR   R5          ; 8A5B  \_ Move to next record. Do not
          B      L_8A55      ; 8A5C  / decrement active record count.

L_8A5E: MVI@   R1,    R3     ; 8A5E  \_ Save opcode byte from breakpoint
          MVO@   R3,    R5     ; 8A5F  /
          MVO@   R2,    R1     ; 8A60  Write SIN opcode in its place
          DECR   R0          ; 8A61  Decrement active count
          B      L_8A51      ; 8A62  Move to next record.
L_8A64: MOVR   R4,    R7     ; 8A64  Leave.
```

\$8A65: Set up SIN2 on the other side of the next instruction (single step)

```
L_8A65: PSHR    R4                ; 8A65  Save return address.

          SDBD                ; 8A66  \_ Addr. of saved PC value.
          MVII    #$85F8, R5  ; 8A67  /

          CLRR   R2          ; 8A6A
          ANDI   #$03FF, R0   ; 8A6B  Clean up opcode
          MOVR   R0,    R1    ; 8A6D  Copy to R1
```

```

    CMPI    #$0004, R1      ; 8A6E \_ Is this a J / JSR?
    BEQ     L_8B48         ; 8A70 / Yes: L_8B48: 3 or more.
                                ;      Reconstruct based on where it
                                ;      jumps to.

    CMPI    #$0001, R1      ; 8A72 \_ Is this SDBD?
    BEQ     L_8BAD         ; 8A74 / Yes: L_8BAD: 4 words if imm.

    ANDI    #$0200, R1      ; 8A76 \_ Internal reference instr?
    BEQ     L_8A88         ; 8A78 / Yes: L_8A88.

    MOVR    R0,    R1      ; 8A7A Copy to R1
    ANDI    #$01C0, R1      ; 8A7B \_ Is this a branch instr?
    BEQ     L_8AE3         ; 8A7D / Yes: L_8AE3

    MOVR    R0,    R1      ; 8A7F Copy to R1
    ANDI    #$0038, R1      ; 8A80 \_ Is this direct mode?
    BEQ     L_8AED         ; 8A82 / Yes: L_8AED

    CMPI    #$0038, R1      ; 8A84 \_ Is this immediate mode?
    BEQ     L_8AED         ; 8A86 / Yes: L_8AED

L_8A88: MOVR    R0,    R3      ; 8A88 \
    ANDI    #$03F8, R3      ; 8A89 |- Is this an implied 1-op?
    BEQ     L_8ADF         ; 8A8B / Yes: L_8ADF: 1 word instr

    MOVR    R0,    R3      ; 8A8D \
    ANDI    #$0007, R3      ; 8A8E |_ Is the destination R7?
    CMPI    #$0007, R3      ; 8A90 | No: L_8ADF: 1 word instr
    BNEQ   L_8ADF         ; 8A92 /

    MOVR    R0,    R3      ; 8A94 \
    ANDI    #$01C0, R3      ; 8A95 |- 2-op Reg-to-Reg, w/ R7 dest?
    BNEQ   L_8AAD         ; 8A97 / Yes: L_8AAD: Need to sim.

    MOVR    R0,    R3      ; 8A99 \
    ANDI    #$0038, R3      ; 8A9A |_ ?? not sure what gets to here.
    CMPI    #$0000, R3      ; 8A9C | Yes: L_8ADF: 1 word instr
    BEQ     L_8ADF         ; 8A9E /

    CMPI    #$0030, R3      ; 8AA0 \_ GSWD?
    BEQ     L_8ADF         ; 8AA2 / Yes: L_8ADF: 1 word instr

    CMPI    #$0038, R3      ; 8AA4 \_ NOP, SIN?
    BEQ     L_8ADF         ; 8AA6 / Yes: L_8ADF: 1 word instr

    MOVR    R0,    R3      ; 8AA8 \_ Mask destination to R0/R1
    ANDI    #$03F9, R3      ; 8AA9 /
    B       L_8ABA         ; 8AAB Need to simulate it...

L_8AAD: CMPI    #$0040, R3      ; 8AAD \_ Is this really a shift?
    BEQ     L_8ADF         ; 8AAF / Yes: It's a 1 word instr.

```

```

    CMPI    #$0140, R3          ; 8AB1 \_ Is this a CMPR xx, R7?
    BEQ     L_8ADF              ; 8AB3 / Yes: That's weird, but 1 word.

;; ----- ;;
;; @Reg-Reg instr that modifies R7; simulate it. ;;
;; ----- ;;
L_8ABA: MOVR    R0,    R3          ; 8AB5
        ANDI    #$03C0, R3       ; 8AB6
        XORI    #$0021, R3       ; 8AB8 Force dest to R1
        SDBD    ; 8ABA \_ R4 => self-mod buffer
        MVII    #$8604, R4       ; 8ABB /
        MVO@    R3,    R4        ; 8ABE Write modified opcode
        MVII    #$00AF, R3       ; 8ABF \_ Write JR R5 opcode
        MVO@    R3,    R4        ; 8AC1 /
        SDBD    ; 8AC2 \
        MVI@    R5,    R3        ; 8AC3 | \_ Set R1 = saved PC + 1
        INCR    R3              ; 8AC4 |
        MOVR    R3,    R1        ; 8AC5 /
        MOVR    R0,    R3        ; 8AC6 \
        ANDI    #$0038, R3       ; 8AC7 |
        SLR     R3,    2         ; 8AC9 |
        SDBD    ; 8ACA | \_ Get saved register arg into R4.
        MVII    #$85EA, R5       ; 8ACB |
        ADDR    R3,    R5        ; 8ACE |
        SDBD    ; 8ACF |
        MVI@    R5,    R4        ; 8AD0 /
        CMPI    #$000C, R3       ; 8AD1 Is reg arg R6?
        BNEQ    L_8AD6          ; 8AD3 No: keep going

L_8AD6: DECR    R4              ; 8AD5 Yes: pre-decrement.
        SDBD    ; 8AD6 \
        MVII    #$85E8, R5       ; 8AD7 |
        SDBD    ; 8ADA | \_ Restore processor flags
        MVI@    R5,    R3        ; 8ADB |
        RSWD    R3              ; 8ADC /
        B       L_8BFD          ; 8ADD Go do it!

;; ----- ;;
;; A single-word instruction ;;
;; ----- ;;
L_8ADF: MVII    #$0001, R1       ; 8ADF
        B       L_8BB6          ; 8AE1

;; ----- ;;
;; An apparent branch. Decode for the details. ;;
;; We know bit 9 = 1, bit 8:6 = 0. ;;
;; ----- ;;
L_8AE3: MOVR    R0,    R1          ; 8AE3 \
        ANDI    #$01D0, R1       ; 8AE4 | \_ Is this maybe BEXT?
        BNEQ    L_8AED          ; 8AE6 / Yes: L_8AED

        MVII    #$0002, R2       ; 8AE8 \

```



```

MOVR    R2,    R1          ; 8AEA  |- 2 word instruction, and need
B       L_8BB6          ; 8AEB  /   to hook branch target too.

;; ----- ;;
;; A likely 2-word instr. Check for exceptions to that rule. ;;
;; ----- ;;
L_8AED: MOVR    R0,    R3          ; 8AED  \
ANDI    #$0007, R3          ; 8AEE  |- Is dest R7?
CMPI    #$0007, R3          ; 8AF0  /
BNEQ   L_8B44          ; 8AF2  No: 2 word instr.

MOVR    R0,    R3          ; 8AF4  \
ANDI    #$03C0, R3          ; 8AF5  |- Is this in Branch space?
CMPI    #$0200, R3          ; 8AF7  | Yes: 2 word instr.
BEQ    L_8B44          ; 8AF9  /

CMPI    #$0240, R3          ; 8AFB  MVO@ R7, xx or MVO R7, xx?
BEQ    L_8B44          ; 8AFD  2 word instr.

CMPI    #$0340, R3          ; 8AFF  CMP@ R7, xx?
BEQ    L_8B44          ; 8B01  2 word instr.

MOVR    R0,    R3          ; 8B03  \
ANDI    #$0038, R3          ; 8B04  |- Direct mode: needs sim
BEQ    L_8B2E          ; 8B06  /

CMPI    #$0001, R1          ; 8B08  \_ I think this is always taken?
BNEQ   L_8B2E          ; 8B0A  /

;; ----- ;;
;; Not sure the following is reachable. It attempts to execute ;;
;; an instruction to see what it might do to the target register. ;;
;; ----- ;;
MOVR    R0,    R3          ; 8B0C  Copy opcode to R3. (for no reason)
ANDI    #$03C0, R0          ; 8B0D  Keep the opcode
XORI    #$0021, R0          ; 8B0F  Src=R4, Dst=R1
SDBD                                ; 8B11
MVII    #$8604, R4          ; 8B12
SDBD                                ; 8B15
MVI@   R5,    R3          ; 8B16  Get saved PC value into R3
MOVR    R3,    R5          ; 8B17  Copy to R5
INCR    R3          ; 8B18  \
INCR    R3          ; 8B19  |- Add 2 to skip instr, and put
MOVR    R3,    R1          ; 8B1A  /   into R1.
MVI@   R5,    R3          ; 8B1B  \_ Copy first word to $8604
MVO@   R3,    R4          ; 8B1C  /
MVO@   R0,    R4          ; 8B1D  Copy modified opcode to $8605
MVII    #$00AF, R3          ; 8B1E  \_ Write JR R5 opcode
MVO@   R3,    R4          ; 8B20  /   out to $8606.
MOVR    R0,    R3          ; 8B21  \
ANDI    #$0038, R3          ; 8B22  |
SLR    R3,    2          ; 8B24  |
SDBD                                ; 8B25  |_ Get source register into R4

```

```

MVII  #85EA, R5          ; 8B26 |
ADDR  R3, R5           ; 8B29 |
SDBD  ; 8B2A |
MVI@  R5, R4          ; 8B2B /
B      L_8BFD         ; 8B2C Go execute instr, result is in R1.
                               Exec. continues at L_8BB9
                               ;
;; ----- ;;
;; Execute an instruction to determine its impact on R7. It will ;;
;; actually be reflected in R1. ;;
;; ----- ;;
L_8B2E: MOVR  R0, R3          ; 8B2E
ANDI  #03F9, R3         ; 8B2F Force dest to R1
SDBD  ; 8B31 \_ Point to self-mod area
MVII  #8604, R4        ; 8B32 /
MVO@  R3, R4          ; 8B35 Write modified opcode to $8604
SDBD  ; 8B36 \_ Get saved PC into R3
MVI@  R5, R3          ; 8B37 /
MOVR  R3, R5          ; 8B38 Copy to R5
INCR  R3              ; 8B39 \
INCR  R3              ; 8B3A |- R1 = saved PC + 2
MOVR  R3, R1         ; 8B3B /
INCR  R5              ; 8B3C R5 = saved PC + 1
MVI@  R5, R3         ; 8B3D R3 = @(saved PC+1); R5 = saved PC+2
MVO@  R3, R4         ; 8B3E copy to $8605
MVII  #00AF, R3      ; 8B3F \_ JR R5 to $8606
MVO@  R3, R4         ; 8B41 /
B      L_8BFD         ; 8B42 Go execute instr; result is in R1.
                               Exec. continues at L_8BB9
                               ;
;; ----- ;;
;; 2-word instruction ;;
;; ----- ;;
L_8B44: MVII  #0002, R1      ; 8B44
B      L_8BB6         ; 8B46
                               ;
;; ----- ;;
;; This is a J or JSR. Compute its length. ;;
;; ----- ;;
L_8B48: MVII  #0003, R1      ; 8B48 3 words by default
MOVR  R1, R2         ; 8B4A Not a conditional branch, either
SDBD  ; 8B4B \
MVI@  R5, R0        ; 8B4C \_ Get saved program address
DECR  R5            ; 8B4D |
DECR  R5            ; 8B4E /
INCR  R0            ; 8B4F \
MOVR  R0, R4        ; 8B50 |
MVI@  R4, R0        ; 8B51 |
ANDI  #00FC, R0     ; 8B52 |
SLL   R0, 2         ; 8B54 \_ Reconstruct the target address
SLL   R0, 2         ; 8B55 | of a JSR or J instruction.
SLL   R0, 2         ; 8B56 |
SLL   R0, 2         ; 8B57 |
MVI@  R4, R1        ; 8B58 |

```

```

ANDI    #$03FF, R1      ; 8B59 /
XORR    R0,    R1      ; 8B5B

SDBD    ; 8B5C \
CMPI    #$1000, R1     ; 8B5D |
BMI     L_8BB9         ; 8B60 |_- Is the address in the EXEC?
SDBD    ; 8B62 |   No: Leave estimate at 3 words.
CMPI    #$2000, R1     ; 8B63 |
BPL     L_8BB9         ; 8B66 /

;; Look for EXEC subroutines that read an argument @R5, so we can
;; correctly figure out how to step over their argument.

SDBD    ; 8B68 \
CMPI    #$1A61, R1     ; 8B69 |- Is it SLEEP?
BEQ     L_8B96         ; 8B6C /   Yes: Set length to 4.

SDBD    ; 8B6E \
CMPI    #$1628, R1     ; 8B6F |- Is it GETVELX?
BEQ     L_8B96         ; 8B72 /   Yes: Set length to 4.
SDBD    ; 8B74 \
CMPI    #$16DB, R1     ; 8B75 |- Is it CLRBIT?
BEQ     L_8B96         ; 8B78 /   Yes: Set length to 4.

SDBD    ; 8B7A \
CMPI    #$16E6, R1     ; 8B7B |- Is it SETBIT?
BEQ     L_8B96         ; 8B7E /   Yes: Set length to 4.

SDBD    ; 8B80 \
CMPI    #$1670, R1     ; 8B81 |- Is it RANGE_CHECK?
BEQ     L_8B9A         ; 8B84 /   Yes: Set length to 5.

SDBD    ; 8B86 \
CMPI    #$187B, R1     ; 8B87 |- Is it PRINT?
BEQ     L_8B9E         ; 8B8A /   Discover length of string.

SDBD    ; 8B8C \
CMPI    #$1871, R1     ; 8B8D |- Is it PRINT_SPC?
BEQ     L_8B9E         ; 8B90 /   Discover length of string.

MVII    #$0003, R1     ; 8B92 \_ Length of this instr is 3.
B       L_8BB6         ; 8B94 /

L_8B96: MVII    #$0004, R1 ; 8B96 \_ Length of this instr is 4.
B       L_8BB6         ; 8B98 /

L_8B9A: MVII    #$0005, R1 ; 8B9A \_ Length of this instr is 5.
B       L_8BB6         ; 8B9C /

L_8B9E: MVII    #$0003, R1 ; 8B9E Base instr. length is 3.
SDBD    ; 8BA0 \
MVI@   R5,    R4      ; 8BA1 |
DECR   R5         ; 8BA2 |- Point R4 at the string.

```

```

    DECR    R5                ; 8BA3  |
    ADDR    R1,      R4      ; 8BA4  /
L_8BA5: MVI@   R4,      R0    ; 8BA5  \
    INCR    R1                ; 8BA6  |_ Scan until we find a NUL,
    CMPI    #$0000, R0      ; 8BA7  |_ incrementing length as we go.
    BNEQ    L_8BA5          ; 8BA9  /
    B       L_8BB6          ; 8BAB

;; ----- ;;
;; SDBD: If immediate mode, it's 4 bytes. Otherwise, compute ;;
;; size normally.                                           ;;
;; ----- ;;
L_8BAD: MOVR   R3,      R0    ; 8BAD  \
    ANDI    #$0038, R3      ; 8BAE  |- Is it SDBD ; XXX@ R7?
    CMPI    #$0038, R3      ; 8BB0  /
    BNEQ    L_8AED          ; 8BB2  No: don't adjust size

    MVII    #$0004, R1      ; 8BB4  4 bytes for SDBD ; xxx@ R7
                                ;      aka. SDBD immediate mode

;; ----- ;;
;; Given the instruction length in R0, compute the address of the ;;
;; next instruction. R5 points to the saved PC.                ;;
;; ----- ;;
L_8BB6: SDBD                ; 8BB6  \
    MVI@   R5,      R0      ; 8BB7  |- Compute address of next instr.
    ADDR   R0,      R1      ; 8BB8  /

;; ----- ;;
;; Put a "other-side-of-single-step" SIN,2 opcode at the other ;;
;; side of a single step.                                       ;;
;; ----- ;;
L_8BB9: SDBD                ; 8BB9  \_ Address of 'catch' side of a
    MVII    #$85FD, R5      ; 8BBA  / single step.
    MVI@   R1,      R3      ; 8BBD  Get opcode.
    MVO@   R3,      R5      ; 8BBE  Save it.
    MVII    #$0037, R3      ; 8BBF  SIN,2 instruction
    MVO@   R3,      R1      ; 8BC1  Overwrite opcode with SIN,2
    MOVR   R1,      R3      ; 8BC2  \
    ANDI    #$00FF, R1      ; 8BC3  |
    MVO@   R1,      R5      ; 8BC5  |_ Write address of instruction.
    SWAP   R3,      1       ; 8BC6  |
    ANDI    #$00FF, R3      ; 8BC7  |
    MVO@   R3,      R5      ; 8BC9  /
    CMPI    #$0002, R2      ; 8BCA  If this is a conditional branch,
                                ;      we need to hook the branch tgt too.
    BNEQ    L_8BF4          ; 8BCC  If not a branch, zero it out.

;; ----- ;;
;; If this was a branch, compute the branch target.           ;;
;; ----- ;;
    SDBD                ; 8BCE  \
    MVII    #$85F8, R5      ; 8BCF  |_ Get the address we're

```

```

SDBD                ; 8BD2 | stepping over.
MVI@   R5,    R4    ; 8BD3 /
MVI@   R4,    R0    ; 8BD4 Branch opcode
MVI@   R4,    R1    ; 8BD5 Branch displacement
ANDI   #$03FF, R1   ; 8BD6 Mask branch disp to 10 bits.
                    ; (This seems unnecessary!)
ANDI   #$0020, R0   ; 8BD8 Bit 5 is the disp. inversion flag
BNEQ   L_8BDF       ; 8BDA Non-zero: 1s compl. displacemt

ADDR   R4,    R1    ; 8BDC Add displacement to addr.
B      L_8BE2       ; 8BDD

L_8BDF: SUBR   R1,    R4    ; 8BDF \ Sub displacement from addr
DECR   R4      ; 8BE0 |- 1s complement style
MOVR   R4,    R1    ; 8BE1 /

;; ----- ;;
;; This is a conditional branch. Hook the branch-taken target too. ;;
;; ----- ;;
L_8BE2: SDBD                ; 8BE2
MVII   #$8600, R5         ; 8BE3
MVI@   R1,    R3         ; 8BE6 Get instruction from branch tgt.
MVO@   R3,    R5         ; 8BE7 Store it to $8600
MVII   #$0037, R3        ; 8BE8 \ Put SIN,2 instr in its place.
MVO@   R3,    R1         ; 8BEA /
MOVR   R1,    R3         ; 8BEB \
ANDI   #$00FF, R1        ; 8BEC |
MVO@   R1,    R5         ; 8BEE | Store br. target addr to
SWAP   R3,    1          ; 8BEF | $8601, $8602.
ANDI   #$00FF, R3        ; 8BF0 |
MVO@   R3,    R5         ; 8BF2 /
PULR   R7              ; 8BF3 Return

;; ----- ;;
;; This was not a conditional branch, so zero out the conditional ;;
;; branch taken 'catch' record. ;;
;; ----- ;;
L_8BF4: SDBD                ; 8BF4 \
MVII   #$8600, R5         ; 8BF5 |
CLRR   R0              ; 8BF8 | Not a conditional branch, so
MVO@   R0,    R5         ; 8BF9 | ignore the branch target.
MVO@   R0,    R5         ; 8BFA |
MVO@   R0,    R5         ; 8BFB /
PULR   R7              ; 8BFC Return

;; ----- ;;
;; Branch to the small self-modifying code bit to see what happens ;;
;; to R7. The register argument gets moved to R4, and the result ;;
;; is forced to R1. ;;
;; ----- ;;
L_8BFD: SDBD                ; 8BFD
MVII   #$8BB9, R5        ; 8BFE
J      G_8604           ; 8C01

```

\$8C04: Restore machine state and prepare to jump into the program

```
L_8C04: SDBD          ; 8C04 \_ Processor state struct.
        MVII   #$85F6, R5 ; 8C05 /
        SDBD          ; 8C08 \_ Restore stack pointer
        MVI@   R5,    R6 ; 8C09 /
        SDBD          ; 8C0A \_ Restore target address and
        MVI@   R5,    R0 ; 8C0B |- push on stack.
        PSHR   R0      ; 8C0C /
        SDBD          ; 8C0D
        MVII   #$85F2, R5 ; 8C0E
        SDBD          ; 8C11
        MVI@   R5,    R0 ; 8C12
        PSHR   R0      ; 8C13 Push the saved value of R4
        PSHR   R4      ; 8C14 Push our return address
        SDBD          ; 8C15
        MVII   #$85E8, R4 ; 8C16
        SDBD          ; 8C19 \
        MVI@   R4,    R0 ; 8C1A |- Restore processor flags
        RSWD   R0      ; 8C1B /
        SDBD          ; 8C1C \_ Restore R0
        MVI@   R4,    R0 ; 8C1D /
        SDBD          ; 8C1E \_ Restore R1
        MVI@   R4,    R1 ; 8C1F /
        SDBD          ; 8C20 \_ Restore R2
        MVI@   R4,    R2 ; 8C21 /
        SDBD          ; 8C22 \_ Restore R3
        MVI@   R4,    R3 ; 8C23 /
        SDBD          ; 8C24 \
        MVII   #$85F4, R4 ; 8C25 \_ Restore R5
        SDBD          ; 8C28 |
        MVI@   R4,    R5 ; 8C29 /
        PULR   R7      ; 8C2A return to caller to complete the
        ;              dispatch.
        ;
```

\$8C2B: Interrupt service routine: Detect breakpoints

```
L_8C2B: EIS          ; 8C2B Seems redundant, unless /INTR
        MOVR   R6,    R3 ; 8C2C \
        SUBI   #$0008, R3 ; 8C2D |- Get interrupted PC into R2
        MVI@   R3,    R2 ; 8C2F /
        SDBD          ; 8C30 \
        MVII   #$85A3, R4 ; 8C31 |
        MVI@   R4,    R0 ; 8C34 |- If $85A3 is $36, then what?
        CMPI   #$0036, R0 ; 8C35 | User hitting ESC?
        BEQ    L_8C63   ; 8C37 /

        MOVR   R2,    R1 ; 8C39 This is dead code.
        DECR   R2      ; 8C3A Rewind PC by one word
        SDBD          ; 8C3B \
        MVII   #$85D0, R4 ; 8C3C |- Point to breakpoint buffer
```

```

MVII    #$0008, R0        ; 8C3F /
L_8C41: SDBD              ; 8C41 \_ Get breakpoint addr
MVI@    R4,    R1        ; 8C42 /
CMPR    R1,    R2        ; 8C43 Does it match our addr?
BEQ     L_8C59          ; 8C44 Yes: Hit a breakpoint!

INCR    R4              ; 8C46 Skip saved opcode
DECR    R0              ; 8C47 \_ Loop over all 8 breakpoints.
BNEQ    L_8C41         ; 8C48 /

SDBD              ; 8C4A \
MVII    #$85FE, R4      ; 8C4B |
SDBD              ; 8C4E |- Is this a single-step?
MVI@    R4,    R1        ; 8C4F |
CMPR    R1,    R2        ; 8C50 /
BEQ     L_8C59          ; 8C51 Yes: Handle the single-step.

INCR    R4              ; 8C53 \
SDBD              ; 8C54 |_ Check other side of single-step
MVI@    R4,    R1        ; 8C55 | (conditional branch fork)
CMPR    R1,    R2        ; 8C56 /
BNEQ    L_8D04         ; 8C57 No: Ordinary interrupt. Yawn.

L_8C59: MVI@    R2,    R0        ; 8C59 Get opcode
CMPI    #$0036, R0      ; 8C5A Is it SIN,1?
BEQ     L_8C62         ; 8C5C Yes: Ordinary breakpoint

CMPI    #$0037, R0      ; 8C5E Is it SIN,2?
BNEQ    L_8D04         ; 8C60 Yes: Far side of a single-step

L_8C62: MVO@    R2,    R3        ; 8C62 This looks like dead code.
;                               ; We read R2 from @R3 above, and
;                               ; now we write it unmodified to @R3

L_8C63: SDBD              ; 8C63 \
MVII    #$85F6, R4      ; 8C64 |
MOVR    R3,    R0        ; 8C67 |
ANDI    #$00FF, R0      ; 8C68 |_ Save R6 of the interrupted
MVO@    R0,    R4        ; 8C6A | program (our R6 minus 8)
SWAP    R3,    1         ; 8C6B |
ANDI    #$00FF, R3      ; 8C6C |
MVO@    R3,    R4        ; 8C6E /

; ; Unwind the stack into our register save area for R5 .. R1
SDBD              ; 8C6F
MVII    #$85F4, R4      ; 8C70 Point to R5 in register save area
MVII    #$0005, R1      ; 8C73 5 registers to copy this way.
L_8C75: PULR    R0        ; 8C75 Pop register value to save
MOVR    R0,    R3        ; 8C76 \
ANDI    #$00FF, R0      ; 8C77 |
MVO@    R0,    R4        ; 8C79 |_ put it in save area
SWAP    R3,    1         ; 8C7A |
ANDI    #$00FF, R3      ; 8C7B |
MVO@    R3,    R4        ; 8C7D /

```

```

SUBI    #$0004, R4      ; 8C7E Move to next register in save area
DECR    R1              ; 8C80 \
CMPI    #$0000, R1     ; 8C81 |- iterate over all 5
BNEQ    L_8C75         ; 8C83 /

L_8C85: SDBD           ; 8C85 \_ R4 points to processor flags
MVII    #$85E8, R4     ; 8C86 / save are
MVII    #$0002, R1     ; 8C89 Save two more registers (SWD, R0)
L_8C8B: PULR          R0 ; 8C8B Pop register value to save
MOVR    R0, R3         ; 8C8C \
ANDI    #$00FF, R0    ; 8C8D |
MVO@    R0, R4        ; 8C8F |- put it in save area
SWAP    R3, 1         ; 8C90 |
ANDI    #$00FF, R3    ; 8C91 |
MVO@    R3, R4        ; 8C93 /
DECR    R1            ; 8C94 \
CMPI    #$0000, R1     ; 8C95 |- Iterate over both SWD and R0.
BNEQ    L_8C8B        ; 8C97 /

SDBD           ; 8C99 \_ Address of saved R7
MVII    #$85F8, R4     ; 8C9A /
PULR    R0            ; 8C9D Address we interrupted (ie. R7)
MOVR    R0, R3         ; 8C9E \
ANDI    #$00FF, R0    ; 8C9F |
MVO@    R0, R4        ; 8CA1 |- Save interrupted R7 to
SWAP    R3, 1         ; 8CA2 | the save area.
ANDI    #$00FF, R3    ; 8CA3 |
MVO@    R3, R4        ; 8CA5 /

SDBD           ; 8CA6 \
MVII    #$85A3, R4     ; 8CA7 | Command byte $36 (SIN,1)?
MVI@    R4, R0        ; 8CAA |- (This means users pressed ESC)
CMPI    #$0036, R0    ; 8CAB | Yes: L_8CF1
BEQ     L_8CF1        ; 8CAD /

MVI@    R2, R0        ; 8CAF \
CMPI    #$0036, R0    ; 8CB0 |- Is interrupted opcode SIN,1?
BEQ     L_8CF1        ; 8CB2 / Yes: L_8CF1

SDBD           ; 8CB4 \
MVII    #$8603, R5     ; 8CB5 |
MVI@    R5, R0        ; 8CB8 |- Keep going after single step
CMPI    #$0000, R0    ; 8CB9 | if $8603 is zero.
BEQ     L_8CD1        ; 8CBB /

JSR     R4, L_8D1F    ; 8CBD Restore opcodes from single-step

```

\$8CC0: Re-enter command dispatch loop; override any command request w/NOP

```

L_8CC0: CLRR    R0      ; 8CC0
SDBD           ; 8CC1
MVII    #$8603, R5     ; 8CC2

```



```

MVO@   R0,    R5      ; 8CC5
MVII   #$000B, R0    ; 8CC6 \ Store a NOP command in the
SDBD   ; 8CC8  |_ command byte before jumping
MVII   #$85A3, R5    ; 8CC9 | to the command dispatch loop.
MVO@   R0,    R5      ; 8CCC / 6502 side watches for this.
EIS    ; 8CCD
J      CMD_DISP      ; 8CCE Jump to command dispatch loop ($887B)

```

\$8CD1: Resume execution after single step

```

L_8CD1: SDBD           ; 8CD1 \
        MVII   #$85FD, R5 ; 8CD2 |
        MVI@   R5,    R0  ; 8CD5 |_ Restore first word of single
        SDBD   ; 8CD6  |_ step saved state.
        MVI@   R5,    R1  ; 8CD7 |
        MVO@   R0,    R1  ; 8CD8 /
        SDBD   ; 8CD9 \
        MVII   #$8600, R5 ; 8CDA |
        MVI@   R5,    R0  ; 8CDD |
        SDBD   ; 8CDE  |_ Restore second word of single
        MVI@   R5,    R1  ; 8CDF |_ step saved state, if any.
        CMPI   #$0000, R1 ; 8CE0 |
        BEQ    L_8CE5    ; 8CE2 |
        MVO@   R0,    R1  ; 8CE4 /

L_8CE5: JSR     R4,    L_8D07 ; 8CE5 Replace breakpoints w/ orig. code
        JSR     R4,    L_8A4A ; 8CE8 Install breakpoints.
        JSR     R4,    L_8C04 ; 8CEB Prepare to return to program

        PULR   R4      ; 8CEE Restore saved R4
        EIS    ; 8CEF Lights on!
        PULR   R7      ; 8CF0 Flying leap!

```

\$8CF1: Capture some state to prepare to single-step over a breakpoint

```

L_8CF1: JSR     R4,    L_8D07 ; 8CF1 Uninstall breakpoints.

;; Read @R2, and copy the value read to $85FA. Copy R2 itself to
;; $85FB, $85FC in little-endian order.
SDBD   ; 8CF4 \
MVII   #$85FA, R5 ; 8CF5 |_ Copy opcode @R2 to $85FA
MVI@   R2,    R0  ; 8CF8 |
MVO@   R0,    R5  ; 8CF9 /
MOVR   R2,    R3  ; 8CFA \
ANDI   #$00FF, R2 ; 8CFB | Copy single-step addr to
MVO@   R2,    R5  ; 8CFD |_ $85FB-$85FC
SWAP   R3,    1   ; 8CFE |
ANDI   #$00FF, R3 ; 8CFF |
MVO@   R3,    R5  ; 8D01 /
B      L_8CC0    ; 8D02 Go to dispatch loop, NOP'ing out
                    ; any pending command.

```

\$8D04: Branch to default EXEC ISR at \$1126

```
L_8D04: J      X_DEF_ISR      ; 8D04  X_DEF_ISR = $1126
```

\$8D07: Replace breakpoints with original opcodes/data

```
L_8D07: SDBD                ; 8D07  \  
      MVII   #$85CF, R5      ; 8D08  |- Number of active breakpoints  
      MVI@   R5,    R0      ; 8D0B  /  
L_8D0C: CMPI   #$0000, R0    ; 8D0C  \  
      BEQ    L_8D1E         ; 8D0E  / active breakpoints.  
  
L_8D10: SDBD                ; 8D10  
      MVI@   R5,    R1      ; 8D11  Get address  
      CMPI   #$0000, R1    ; 8D12  \  
      BNEQ   L_8D19        ; 8D14  / address is non-NULL.  
  
      INCR   R5            ; 8D16  \  
      B     L_8D10         ; 8D17  / Otherwise, move to next record.  
  
L_8D19: MVI@   R5,    R3      ; 8D19  \  
      MVO@   R3,    R1      ; 8D1A  / Copy opcode back to memory.  
      DECR   R0            ; 8D1B  \  
      B     L_8D0C        ; 8D1C  / Move to next breakpoint slot.  
L_8D1E: MOVR  R4,    R7      ; 8D1E  Return.
```

\$8D1F: Restore opcodes after a single step

```
L_8D1F: SDBD                ; 8D1F  
      MVII   #$85FD, R5      ; 8D20  
      MVI@   R5,    R0      ; 8D23  Read opcode to restore.  
      SDBD                ; 8D24  
      MVI@   R5,    R1      ; 8D25  Read address  
      CMPI   #$0000, R1    ; 8D26  Was it zero?  
      BEQ    L_8D2B        ; 8D28  Yes: skip  
  
      MVO@   R0,    R1      ; 8D2A  No: restore the opcode.  
  
      ; Conditional branches store two opcodes: One for each side of  
      ; the fork. Restore the second opcode if it was saved.  
L_8D2B: MVI@   R5,    R0      ; 8D2B  Read second opcode.  
      SDBD                ; 8D2C  
      MVI@   R5,    R1      ; 8D2D  Read second address  
      CMPI   #$0000, R1    ; 8D2E  Was it zero?  
      BEQ    L_8D33        ; 8D30  Yes: skip  
  
      MVO@   R0,    R1      ; 8D32  No: restore the opcode.  
L_8D33: MOVR  R4,    R7      ; 8D33  Return
```

What we still don't know

While we know quite a lot about the Black Whale now, there's still some details we're lacking:

- What sort of terminal / terminal software was used to talk to the Black Whale?
- Did the Black Whale work with both NTSC & PAL systems, or only NTSC?
- Was a T-Card required for RAM at \$5000 - \$6FFF, or were some KC's modified?
 - A number of 1983-era prototypes have \$5000 - \$6FFF, \$9000 - \$AFFF memory maps, suggesting that \$A000 - \$BFFF was not remapped to \$5000 - \$6FFF.
- What did versions 1 through 4 of the software look like? The listing analyzed above is version 5.
- Is there a version after version 5?

Anything else?

Some tantalizing hearsay

The [interview with Patrick Aubry](#) includes this photograph:



That terminal appears to be a [C. ITOH CIT101 VT-100 clone](#). That terminal sports [two bidirectional serial ports and the ability to bridge between them](#). That seems ideal for this particular use-case.

In particular, its voltages are a good match:

Output RS232-C Levels: Mark level -6.0 V to -12.0 V. Space level 6.0 V to 12.0 V.

Input RS232-C Levels: Mark level -25.0 V to 0.75 V or open circuit. Space level 2.25 V to 25.0 V.

And, it supports flexible bridging between both its serial ports:

2.6.5.4 Bidirectional Auxiliary Port Protocol. The CIT-101 Bidirectional Auxiliary Port is a powerful extension of the printer output port found on some terminals and may be used as a simple output port to drive a local printer. In this mode the terminal can accept VT52 and VT100 printer commands. Additional support for this mode is provided by a single wire READY/BUSY handshake protocol which may be used instead of, (or in addition to), the XON/XOFF handshake. For more sophisticated applications, the Bidirectional Auxiliary Port may be used for both input and output and may be interchanged with the main Communications Port in many situations. Data may be directed from the Keyboard to either the COM Port or the AUX Port, from the COM Port to either the Display or the AUX Port or both, and from the AUX Port to either the Display or the COM Port or both. To provide even greater flexibility, these paths may all be selected and controlled by either the Keyboard or the host software.

Its Non-volatile RAM dump bears a strong resemblance to the pseudo-hex dumps from the Black Whale:

`'CHECKSUM ERROR ON NVR' [parameter]`

Indicates a fault in the Non-Volatile RAM (NVR) circuitry detected by comparing the checksum stored in NVR at the time of the last SAVE operation with the checksum computed by the diagnostic. May mean that one or more SET-UP bits or features may be altered or unstable. The parameter given is two ASCII characters representing two 4-bit parts of the 8-bit checksum computed, where each part has been added to hex 40 for the purposes of display (for example, @@ is 00, @A is 01, etc.). Perform a SAVE operation to clear.

The protocol between the Black Whale and the serial port does some strange things with the MSB. I could not find anything interesting in the C. Itoh CIT101 docs regarding the MSB, however. That detail remains open.

The TV in the back appears to be a [Hitachi 13" TV from circa 1983](#). Do PAL/SECAM TVs require the dual tuning knobs, or is this a distinct NTSC-ism from North America? That would tell us whether the system in the background is NTSC or PAL. (It looks like NTSC due to the fact the screen is filled out 100%.)



VISUAL
ALCHEMY, LLC
video & computer services for film production