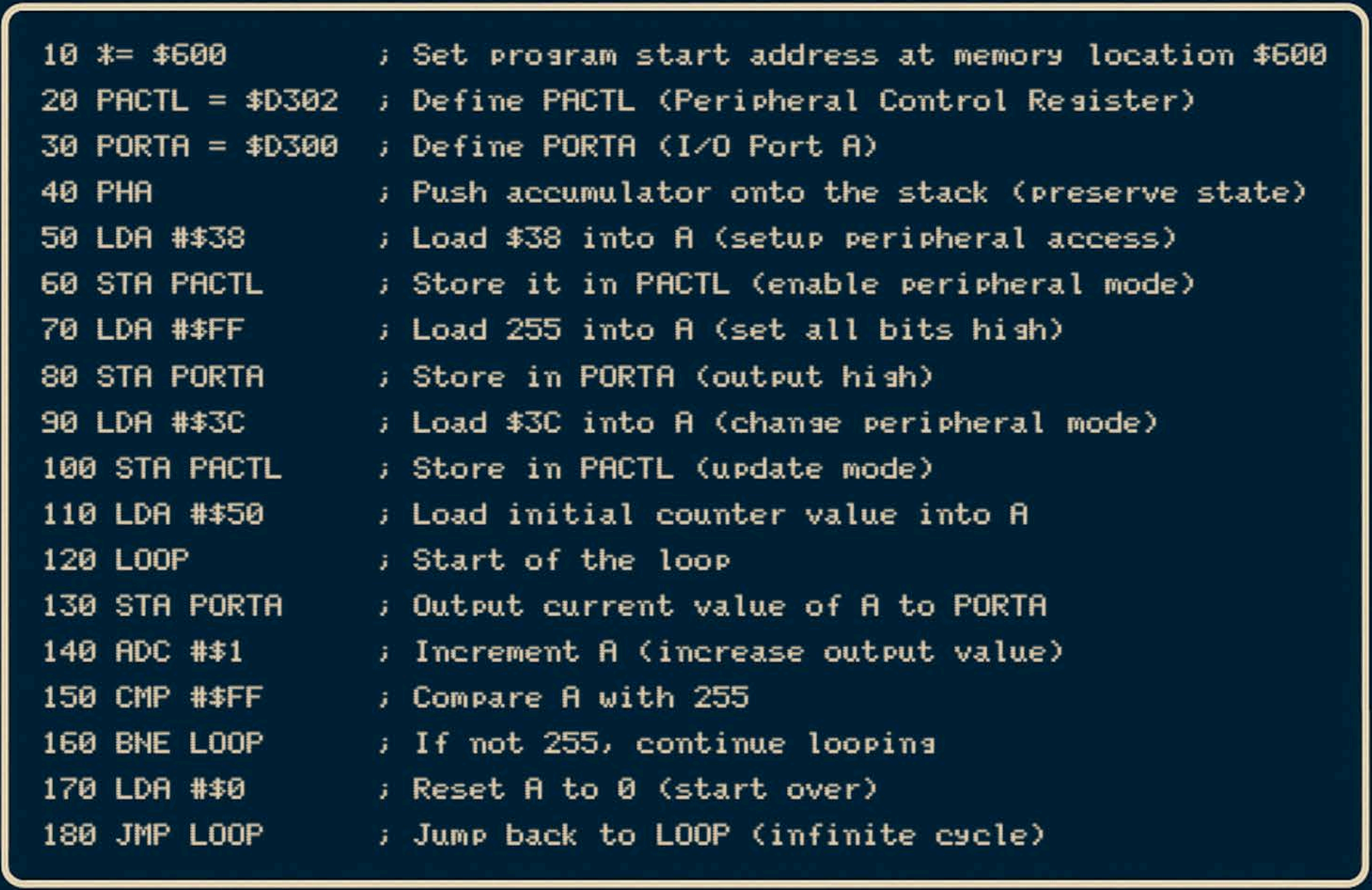
Altirra Debugger Help



```
10 *= $600          ; Set program start address at memory location $600
20 PACTL = $D302    ; Define PACTL (Peripheral Control Register)
30 PORTA = $D300    ; Define PORTA (I/O Port A)
40 PHA              ; Push accumulator onto the stack (preserve state)
50 LDA #$38         ; Load $38 into A (setup peripheral access)
60 STA PACTL        ; Store it in PACTL (enable peripheral mode)
70 LDA #$FF         ; Load 255 into A (set all bits high)
80 STA PORTA        ; Store in PORTA (output high)
90 LDA #$3C         ; Load $3C into A (change peripheral mode)
100 STA PACTL       ; Store in PACTL (update mode)
110 LDA #$50        ; Load initial counter value into A
120 LOOP            ; Start of the loop
130 STA PORTA       ; Output current value of A to PORTA
140 ADC #$1         ; Increment A (increase output value)
150 CMP #$FF        ; Compare A with 255
160 BNE LOOP        ; If not 255, continue looping
170 LDA #$0         ; Reset A to 0 (start over)
180 JMP LOOP        ; Jump back to LOOP (infinite cycle)
```

Avery Lee

Altirra 4.31 Debugger Help

This help is generated from Altirra by the **Help > Debugger Help** command on the menu.

It can also be accessed contextually in the debugger itself using the **.help** command.

Table of Contents

Common usage

To enter the Altirra debugger: **Debug > Enable debugger**

```
Commands <xaddress> accept extended addresses:
$0000 CPU view of primary memory
$01:0000 CPU view, 65C816 high memory
$EF'4000 CPU view, extended memory ($4000 with PORTB=$EF)
n:$0000 ANTIC view of primary memory
v:$00000 VBXE memory
r:$0000 Main memory
rom:$0000 System ROM (OS, BASIC, self-test)
x:$00000 Extended memory
cart:$0000 Cartridge ROM, linear view
t:$01'A000 Cartridge memory, banked view (bank $01, address $A000)

Some commands support length syntax:
db siov L100
db 4000 L>5FFF

Commands taking <path> also accept ? to interactively select the file.

Use .help <command> for detailed help on that command.
```

` Bypass aliases

```
Executes a command without checking the alias table.

`<command>

Allows access to commands that are blocked by aliases. This also
provides an
escape hatch if you somehow manage to alias over both the alias set (as)
and
alias clear (ac) commands.
```

~ Target control

```
Switches or displays information about debugging targets.

~ (Display target list)
~0s, ~1s, ... (Switch current target)

The default debugging target is target 0, which is the main computer
CPU.
When a device that has a coprocessor is present, such as the Veronica
cartridge, additional targets will be available.

Not all commands work with all targets; many are restricted to target 0.
```

? Evaluate expression

```
Prints the value of an expression.

? <expression>

The expression may contain the following elements, from highest to
lowest
precedence:

Constants
iii Integer
$hhh Hex integer
%bbbb Binary integer
ii:iiii Banked address (decimal)
$hh:hhhh Banked address (hex)

Variables
pc PC register
x X register
y Y register
s S register
p P register
hpos ANTIC horizontal position counter
vpos ANTIC vertical position counter
address PC or memory address from current breakpoint
value Value written from current access write breakpoint
xbankreg Bank register value for PORTB extended memory
xbankcpu PORTB extended memory bank (CPU)
xbankantic PORTB extended memory bank (ANTIC)
@t0-t9 Temporaries
@ra Return address (if RTS executed immediately)
@frame ANTIC frame counter
@clk Clock cycle counter
@cclk CPU clock cycle counter
@xpc Extended PC (banked memory)
@tapepos Tape position in samples

Functions
@hwwritereg(addr)
Read back ANTIC, POKEY, or GTIA write-only registers
```

Grouping
(exp) Subexpression
@(...) Immediately evaluated subexpression

Unary operators
db addr Read memory unsigned byte
dsb addr Read memory signed byte
dw addr Read memory unsigned word
dsw addr Read memory signed word
dsd addr Read memory signed double word (32-bit)
+ Unary plus
- Unary minus
< Unary low byte (bits 0-7, as unsigned byte)
> Unary high byte (bits 8-15, as unsigned byte)
! Logical negation

Multiplicative operators
* Multiply
/ Integer division (truncation toward zero)
% Modulus (a%b = a - a/b*b)

Additive operators
+ Add
- Subtract

Relational operators
< Less-than
<= Less-equal
> Greater-than
>= Greater-equal
= Equal
!= Not equal

Bitwise operators 1
& Bitwise AND

Bitwise operators 2
| Bitwise OR
^ Bitwise XOR

Logical operators 1
and Logical AND

Logical operators 2
or Logical OR

Ternary operators
?: Conditional operator

Spaces are not significant except to delimit tokens.

Symbol names may also be used in expressions, but care should be taken to distinguish between "symbol", which evaluates to the symbol's address,
and "db symbol" or "dw symbol", which reads a byte or word at the address.
Symbols that conflict with other tokens can be forced by prefixing with #, i.e. #address. A symbol from a specific module may be referenced with the syntax modname!symname.

All expressions are evaluated with 32-bit signed two's-complement integer arithmetic. Logical operators operate according to the rules of the C programming language, where 0 signifies false, a non-zero value is interpreted as true, and true is encoded as 1.

Due to a conflict with the bank operator, the ternary operator may sometimes require a space between a constant and the ':' in order to be parsed as intended.

Examples:

```
vdslst Address of DLI vector ($0200)
dw vdslst DLI routine address
dw vdslst+2 DLI routine address + 2
dw vdslst + 2 DLI routine address + 2
dw (vdslst+2) SIO proceed IRQ routine address
```

a Assemble

Invoke 6502 miniassembler.

```
a <address>
```

Allows entry of 6502 assembly language code starting at a particular address. Enter one line at a time, and enter a blank line to end.

Labels may be entered and referenced; created labels are entered into the custom symbol table. Labels must be defined before use and forward references are not allowed.

Auto-increment syntax is allowed: LDA $1000,X+ -> LDA $1000,X / INX

Statements may be stacked to share an operand: LDA:RNE VCOUNT

Pseudoinstruction macros supported:

```
MVA src dst -> LDA src / STA dst
MVX src dst -> LDX src / STX dst
MVY src dst -> LDY src / STY dst
MWA src dst -> Move 16-bit data using LDA/STA
MWX src dst -> Move 16-bit data using LDX/STX
MWY src dst -> Move 16-bit data using LDY/STY
Rcc -> Bcc *
Scc:<insn> -> Bcc next / <insn> / next:
Jcc -> branch + jump
INW/DEW -> INC/DEC / BNE / INC/DEC
INL/DEL -> INC/DEC 24-bit "long" value
IND/DED -> INC/DEC 32-bit "double-word" value
ADD -> CLC+ADC
SUB -> SEC+SBC
ADB/SBB src1 src2 dst -> Store src1-src2 in dst (8-bit)
ADW/SBW src1 src2 dst -> Store src1-src2 in dst (16-bit)
CPB/CPW/CPL/CPD src1 src2 -> Compare src1-src2 (8/16/24/32-bit)
PHR -> Push A, X, Y
PLR -> Pop Y, X, A
```

Supported directives:

```
ORG address Shift code origin to new address
```

a8 Set Atari800-compatible command aliases

Sets compatibility aliases for commands from the Atari800 emulator debugger.

```
a8
```

The following commands are mapped:

```
cont g Continue (go)
show r Show registers
stack k Show stack
setpc * r pc * Change PC register
seta * r a * Change A register
setx * r x * Change X register
sety * r y * Change Y register
sets * r s * Change S register
setn [*] r p.n */1 Set N flag
setv [*] r p.v */1 Set V flag
setd [*] r p.d */1 Set D flag
seti [*] r p.i */1 Set I flag
setz [*] r p.z */1 Set Z flag
setc [*] r p.c */1 Set C flag
clrn r p.n 0 Clear N flag
clrv r p.v 0 Clear V flag
clrd r p.d 0 Clear D flag
clri r p.i 0 Clear I flag
clrz r p.z 0 Clear Z flag
clrc r p.c 0 Clear C flag
c e Change (enter)
d u Disassemble (unassemble)
f * * *.. f * L>* *.. Fill memory
m * [*] db * [L>*] Memory list (display bytes)
s * *.. s * [L>*].. Search memory
sum * * .sum * L>* Compute memory sum
bpc bp Set PC breakpoint
history h Show CPU history
g t Go one insn (trace)
r gr Execute until return
b bl/bc/bp Breakpoints
antic .antic Display ANTIC status
gtia .gtia Display GTIA status
pia .pia Display PIA status
pokey .pokey Display POKEY status
dlist .dumpdlist Show display list
labels .loadsym Load symbols
coldstart .restart Cold-reset the simulation
warmstart .warmreset Warm-reset the simulation
help .help Display command help
```

ac, al, as Manage command aliases

```
Set, clear, or list aliases for debugger commands.

ac (Clear all aliases)
al (List aliases)
as <alias> <command> (Set an alias)
as <alias> (Unset an alias)

Allows you to set alternate name aliases for commands. If an alias is
set
with the same name as a standard debugger command, the alias takes
precedence. An alias always redirects to a standard command; an alias
cannot point to another alias.

To access a command that is blocked by an alias, prefix the command name
with a backtick (`).
```

ap Add command alias pattern

```
Adds a command alias pattern for a debugger command.

ap "pattern" (Clear alias pattern)
ap "pattern" "alias" (Set alias pattern)

Allows a command alias to be set that matches and transforms arguments.
The pattern string matches a command and optional arguments, while the
alias string determines the resulting command and arguments.

Tokens %0, %1, %2... up through %9 can be used to match and substitute
parameters. They must appear as or at the end of an argument in the
pattern. The special token %* can also be used to capture and recall
a variable number of remainder of arguments (varargs). It can only
capture and recall entire parameters.

If multiple alias patterns are declared for the same command, they are
added and tested in order and the first one to match is used. Therefore,
patterns should be added from least to most general.

Examples:

ap "foo" "bar"

Matches the command "foo" with no arguments and executes "bar"
in response.

ap "foo x" "bar y"

Matches the command "foo" with a single argument "x" and executes
the command "bar y" in response.

ap "foo %1 %2 %*" "bar %1 L>%2 %*"

Matches the command "foo" with two or more arguments and executes
the command "bar" with the second argument prefixed with L>.


ap "foo %1 L%2" "bar %1 %2"

Matches the command "foo" with exactly two arguments and the second
argument prefixed with L, and executes "bar" with the second
argument stripped of its prefix.

See also: ac, al, as
```

ba Break on memory access

Set or clear a memory access breakpoint.

```
ba [-n] [-k] [-q] [-g <group>] r <address> [L<length>] ["cmd"]
(Break on memory read)
ba [-n] [-k] [-q] [-g <group>] w <address> [L<length>] ["cmd"]
(Break on memory write)
ba r/w * (Clear memory read/write access breakpoints)
```

Memory access breakpoints cause the emulator to stop whenever the CPU core reads or writes a particular address. There can be an unlimited number of both read and write breakpoints. If a length greater than 1 byte is specified, a range breakpoint is created.

If a quoted string is supplied, it is run as a command whenever the breakpoint is hit.

DMA accesses, such as by ANTIC or VBXE, do not trip access breakpoints. All breakpoints are by CPU address and trip regardless of any bankswitching or other memory overlays.

Memory read breakpoints can be tripped by false reads from the CPU core. This happens for certain indexed operations. Although generally invisible
to normal program operation, these false reads are real and can cause malfunctions with memory mapped hardware, such as cartridge banking registers that are sensitive to reads. False writes are rarer but can occur with read/modify/write instructions.

The -n, -k, -q, and -g options are the same as for the bp command.

Using * as the address will clear all access breakpoints of the indicated type.

See also: bp (set breakpoint)

bc Clear breakpoint(s)

Clear PC, memory, or expression breakpoints.

```
bc <index> (Clear a breakpoint)
bc * (Clear all breakpoints)
```

Removes a breakpoint by index, or if * is specified, all breakpoints. This works for any breakpoint listed by the bl (breakpoint list) command.

By default 'bc' only clears breakpoints in the main breakpoint group. Breakpoints in other groups can be cleared by prefixing with with the group name in a period, e.g. foo.1. Similarly, an entire group can be cleared with 'bc group.*'.

See also: ba, bp, bl, bx

bl List breakpoints

Lists all currently set breakpoints.

```
bl [-a] [-v] [-t] (List breakpoints)
```

Displays a list of all breakpoints. The displayed breakpoint indices can be used in other commands such as bc (breakpoint clear).

Flags displayed on each breakpoint:

K - auto-clear on reset
O - one-shot
PC - break on CPU PC address
R - break on read access
W - break on write access

Deferred breakpoints are also listed and are ones that haven't been bound to an actual address yet. These remain inactive until symbols have been loaded that contain an entry for the source file/line reference.

-a lists breakpoints in all groups. Normally only breakpoints in the main group are listed.

-t lists breakpoints for all targets, instead of just the current target.

-v shows verbose information about internal system and CPU core breakpoints.

See also: bc, bl, bp, bx

bp Set breakpoint

Set a breakpoint at a PC address.

```
bp [-n] [-k] [-q] [-h] [-g group] <xaddress> ["command"]
bp [-n] [-k] [-q] [-h] [-g group] `file:line` ["command"]
```

Sets a PC breakpoint in the CPU core, which cause the emulator to stop when the PC is equal to the address of a breakpoint. There are no limits to the number of PC breakpoints. Breakpoints do not modify memory and work even after the location has been modified; they also fire at an address regardless of any bank switching.

The address expression is evaluated when the bp command is executed, not when the breakpoint is tested. "pc+3", for instance, sets the breakpoint at 3 bytes after the position that the program counter is at now.

The -n option creates a non-stopping breakpoint after which execution continues.

Breakpoints created with -k are automatically cleared on reset.

-g creates the breakpoint in a named group. Named groups have a separate numbering than the main user group and are not visible by default.

-q suppresses normal output of the command.

-h marks the breakpoint as hidden, suppressing UI updates. This is useful
for temporary breakpoints set automatically, such as tracepoints.

If a command is specified, the command is executed when the breakpoint fires. Multiple commands may be specified using ; as a separator.
The normal breakpoint hit message is suppressed, but execution will stop after the commands are executed unless a command is also issued to continue execution. This can be used to implement tracepoints.

Examples:

```
bp -n pc "? db(rtclok+2)"
```

Set a breakpoint at the current location that prints the low byte of the OS real-time clock, and continue execution (-n).

```
bp -n main "r @t1 @frame-@t0; r @t0 @frame"
```

Set a non-blocking breakpoint on main so that whenever main is hit, temporary variable @t1 is set to the difference in the frame counter since the last time (@t0) and update @t0 to the current frame counter value.

Breakpoints can also be set on source lines using source notation:

```
bp `library.s:207`
```

The breakpoint is set immediately if symbols are available. Otherwise, a deferred breakpoint is created and the breakpoint is bound to the code address when the symbols are loaded.

If an extended memory address is used, a bank-sensitive breakpoint is created that only triggers if the PORTB banking configuration matches. For instance, setting a breakpoint on $EF'41AC will trigger when PC=$41AC and PORTB=$EF.

See also: ba, bs

bsc Set breakpoint condition

Set the condition expression on a breakpoint.

```
bsc <id> <condition>
```

Sets or changes the conditional expression that determines whether a breakpoint is hit. For instance, if breakpoint 0 at $4000 is firing too often when called with a return address of $3980, that case can be filtered out as follows:

```
bsc 0 "@ra != $3980"
```

The conditional expression on a breakpoint may differ from the one supplied to the 'bx' command due to extraction of the PC or access subexpression. Use the bl (list breakpoints) command to check the actual condition.

See also: bc (clear breakpoints), bl (list breakpoints)

bt Set PC breakpoint with trace (tracepoint)

Set a tracepoint at a PC address.

```
bt [-k] [-q] [-h] [-g group] { <address> | `file:line` }
format [arguments...]

bt [-k] [-q] [-h] [-g group] { <address> | `file:line` }
format [arguments...] -- [return-arguments...]
```

Sets a non-stopping breakpoint at a PC address or source location with a command to print a trace line on the console when the breakpoint is hit (tracepoint). The breakpoint location is the same as for the bp (set breakpoint) command and the formatting arguments are the same as for the .printf command.

The -k, -q, -g, and -h options work the same as for the bp command.

This command is equivalent to using the bp -n command with a breakpoint command of the form:

```
`.printf <format> <arguments...>
```

The @ts variable can be used to substitute the output of the last .sprintf command as the format string.

The second form of the command with '--' allows a return tracepoint to be set. The format string is evaluated with the first set of arguments prior to -- when the tracepoint is first hit. A second, one-time tracepoint is then set at the return address (@ra) with the resulting string as the format, expanded by the second set of arguments.

Examples:

```
bt siov "SIOV command=$%02X aux=$%04X" db(dcomnd) dw(daux1)
bt fadd "FADD(%g+%g) = %%g --> $%04X" fr0 fr1 @ra -- fr0
```

See also: bp, .printf, .sprintf

bta Set access breakpoint with trace (tracepoint)

Set a tracepoint on access to a memory location or range.

```
bta [-k] [-q] [-h] [-g group] <mode> <address> [L<length>]
[format [arguments...]]
```

Sets a non-stopping breakpoint at a memory location or range with a command to print a trace line on the console when the breakpoint is hit (tracepoint). The breakpoint location is the same as for the ba (break on memory access) command and the formatting arguments are the same as for the .printf command.

The -k, -q, -g, and -h options work the same as for the bp command.

This command is equivalent to using the ba -n command with a breakpoint command of the form:

```
`.printf <format> <arguments...>
```

The @ts variable can be used to substitute the output of the last .sprintf command as the format string.

If no format string is specified, a default trace message is provided.

Examples:

```
bta w skctl
bta r prior "PRIOR=$%02X" value
bta w D100 L100 "[$%04X]=$%02X" value
```

See also: bt, .printf, .sprintf

bx Break on expression (conditional breakpoint)

Set a breakpoint conditional on an expression.

```
bx [-k] [-q] [-n] [-g group] <expression> ["command"]
```

Creates a breakpoint that only activates when the expression is true. In addition to the normal expression elements, conditional breakpoints may have the following additional clauses:

```
read=<address> Read access to address
write=<address> Write access to address
read >[=] <lo> and read <[=] <hi> Read access to address range
write >[=] <lo> and write <[=] <hi> Write access to address range
```

The debugger recognizes and optimizes for three specific patterns:

1) A single PC address.
2) A single read or write address.
3) A read or write address range.

When these patterns are recognized, the appropriate breakpoint type is created and any remainder of the expression is set as the conditional (filter) expression on the breakpoint. Breakpoints not matching any of the above patterns are created as per-instruction breakpoints, executed before each instruction; these are powerful but slow.

The debugger can optimize some conditional expressions for faster evaluation, such as removing redundant logical negation (!) operators and folding constants. An error is reported if the conditional expression is determined to always evaluate to true or false.

If the expression reduces to a single PC or access check, the result is the same as the equivalent bp or ba command.

Examples:

```
bx "pc=$4000 and y=4"
bx "write=rtclok+2 and db(rtclok+2)=$04"
bx "write=rtclok+2 and pc>$c000"
bx "write = consol and value != $08"
bx "read<$fffa and db(pc)!=$20 and db(pc)!=$60 and read>=$d800"
bx db(pc)=$48
```

If a command is specified, it is executed when the breakpoint trips; see the bp (breakpoint) command for details.

Example:

```
bx "pc=ciov and x=$10" "r; g"
```

The -k, -q, and -g options work the same as for the bp command.

See also: ? (evaluate expression), bp (breakpoint)

bs Break on disk sector

Set a breakpoint when D1: reads a particular virtual disk sector.

```
bs <sector> (Set disk sector breakpoint)
bs * (Clear disk sector breakpoint)
```

Stops emulation when the first disk drive (D1:) receives a request to read a particular virtual disk sector.

c Compare memory

Compare two blocks of memory for differences.

```
c <xaddress> L<length> <xaddress2>
```

The two blocks of memory are compared byte-by-byte, and any differences reported. Extended addresses can be used; for instance, this command will
compare a RAM copy of the OS against the ROM OS:

```
c E000 L2000 rom:$E000
```

See also: m (move memory), .sum (compute memory sum), .crc (compute CRC)

da, db, dbi, dd, df, di, dw Display memory

Shows the contents of memory with a particular interpretation.

```
da [<xaddress> [L<length>]] (Display ATASCII string)
db [<xaddress> [L<length>]] (Display bytes)
dbi [<xaddress> [L<length>]] (Display bytes with INTERNAL text)
dd [<xaddress> [L<length>]] (Display 32-bit double words)
df [<xaddress> [L<length>]] (Display decimal float)
di [<xaddress> [L<length>]] (Display INTERNAL string)
dw [<xaddress> [L<length>]] (Display 16-bit words)
```

Length specifies the number of elements to display.

The display commands read memory in debug mode and do not trigger side effects that would normally be caused by a CPU read, such as clearing PIA interrupts or switching cartridge banks.

If no address or length is given, memory dumping continues from the end of the last continuable operation.

The db and dbi commands take additional optional switches:

-c:
Changes the text dump to interpret the data as it would be Displayed on a 20 column text mode display, with the reduced 64 byte character set.

-w <width>:
Displays data with <width> bytes per row.

dbx Display bytes from expression

Evaluates an expression with sequential indices and displays the result as a memory dump.

```
dbx [-w <width>] <expr> [L<length>]
```

The expression is evaluated up to <length> times (default 128) with the current 0-based index as 'address', and the result displayed as bytes.

Example:

```
;Display bytes at $4000 after XOR with $AA
dbx "db($4000+address) ^ $aa" L$100
```

dy Display binary

Shows the contents of memory as binary.

```
dy [-c <chars>] [<xaddress> [L<length>]]
```

-c: set '1' bit character, or both 0/1 bit chars

e, eb, ew — Enter (alter) data in memory

Modifies a block of memory with new data.

```
e <xaddress> <expr> [<expr>...] (Enter bytes)
eb <xaddress> <expr> [<expr>...] (Enter bytes)
ew <xaddress> <expr> [<expr>...] (Enter words)
```

Each successive byte is set to the value of an expression. For eb, expressions must evaluate to unsigned byte range (0-255); use the low byte (<) operator if necessary to forcibly truncate values. For ew, values must be in word range ($0000-FFFF).

Writes by the enter commands have the same effect as CPU writes, including triggering side effects when hardware registers are accessed.

The 'e' command is a synonym for 'eb'.

See also: f (fill memory)

f — Fill memory

Fills a block of memory with a pattern.

```
f <xaddress> L<length> <value>...
```

Each successive byte is set to the value of an expression until <length> bytes have been filled. Expressions must evaluate to unsigned byte range (0-255); use the low byte (<) operator if necessary to forcibly truncate values. The pattern repeats until the specified length is filled.

Examples:

```
f 2000 L1000 aa 55
f 2000 L400 8d <serout >serout
f $01:0000 L10000 0
```

See also: e (enter), fbx (fill bytes with expression)

fbx Fill bytes with expression

Fills a block of memory by evaluating an expression for each location.

```
fbx <xaddress> L<length> <expression>
```

The expression is evaluated once for each byte in the range and its low byte is used to set the value of that location. The VALUE expression token evaluates to the current zero-based index in the range (0-255 only)
and the WRITE expression token gives the current extended address.

Examples:

```
fbx 8000 L1000 value
fbx 8000 L1000 <write
fbx 8000 L1000 "(value-value%80) + (value%40)*2 + (value%80)/40"
```

See also: f (fill)

g Go

Resume execution.

```
g [-n] [-s]
```

By default, the g (Go) command disables source mode debugging. The -n switch preserves the current mode and the -s switch forces source mode.

See also: .sourcemode

gcr Go cycle relative

Resume execution for the given number of machine cycles.

```
gcr <count>
```

The number of cycles can include cycles where the CPU is halted.

gf Go until frame end

Resume execution until the end of the current frame.

```
gf
```

Resumes execution until the beginning of scan line 0 of the next frame. If the beam position is currently at the beginning of scan line 0, an entire frame is executed.

gr Go until return (step out)

Resume execution until the current function is exited.

```
gr
```

Resumes execution until the stack pointer rises above the level of the current function. Any child functions executed due to a JSR instruction or an interrupt are run in their entirety.

gs Go until scanline

Resume execution until the beginning of a particular scanline.

```
gs <scanline>
```

If the current scanline is specified, an entire frame is executed.

gt Go with tracing enabled

Resume execution with per-instruction tracing enabled.

```
gt
```

During traced execution, the current beam location, PC, registers, and instruction are logged.

gv Go until vertical blank interrupt (VBI)

Resume execution until the beginning of the vertical blank interrupt (VBI) is hit.

```
gv
```

If the VBI is disabled, execution stops at the beginning of scan line 248.

h Show CPU history

```
Display history of CPU instructions executed.

h [-i] [-c] [-s <startidx>] [<count> [<pattern>]]

-i: Show only interrupt routine execution
-c: Compress loops
-s: Start from position
<count>: Number of entries to display
<pattern>: Glob wildcard (?*) to search for

The "record instruction history" setting must be enabled in CPU options
for the history command to work.

The history UI pane contains a more powerful history preprocessor that
is
also able to identify subroutines and is recommended for serious history
review.
```

hma Show heat map accesses

```
Shows read and write accesses to memory.

hma [<address> [L<length>]]

The requested memory range is broken down into contiguous ranges that
have been read and written by the CPU.

Note that accelerated accesses from high-level emulation (HLE) and from
ANTIC are not shown in the heat map.

See also: hmc (clear heat map)
```

hmc Clear heat map

```
Clears all tracking data in the heat map.

hmc

All registers and memory locations are marked as preset.

See also: hme (heat map enable)
```

hmd Dump heat map memory status

Dumps the tracking status of memory locations in the heat map.

```
hmd <address> [L<length>]
```

The status of a memory location may be one of the following:

Unknown: Source of data was not tracked.
Preset: Source was not modified since the last clear.
Immediate: Source is the argument of an immediate-mode insn.
Computed: Source is from computation on the CPU.

Data sources are tracked through load/store instructions and through the registers such that the original load source is propagated instead of showing the copy routine itself as the data source.

See also: hmc (clear heat map), hme (heat map enable)

hme Enable or disable heat map

Enables or disables heat map tracing.

```
hme on (Enable heat map)
hme off (Disable heat map)
```

The heat map tracks the source of data as it flows through registers and memory, as well as any read or write accesses to memory.

See also: hma (show heat map accesses), hmc (heat map clear),
hmd (dump heat map memory status),
hmr (heat map register status).

hmp Preset heat map range

Marks a range of memory in the heat map as valid with a preset value.

```
hmp <xaddr> L<length>
```

See also: hmu (unset heat map range)

hmr Show heat map register status

Dump CPU register status being tracked by heat map.

```
hmr
```

The sources of the data in the A, X, and Y registers is shown if known. Additionally, validity information is shown for the A, X, Y, and P registers, indicating which data bits in each register are sourced from valid data and which are based on uninitialized data.

See also: hmc (heat map clear), hme (heat map enable)

hmt View/modify heat map trap options

Views or modifies which access types detected by the heat map can cause debugger traps.

```
hmt (View trap options)
hmt <type> off|early|on (Set mode for single trap type)
hmt * off|early|on (Set mode for all trap types)
```

The heat map engine can detect when uninitialized data is used by 6502 code and force a trap into the debugger when this occurs. The hmt command is used to configure the various trap types:

load:
Activates when uninitialized data is first loaded into the CPU from memory. This traps at the earliest sign of uninitialized memory use, but it can give false positives for some valid cases where uninited memory is merely being copied.

compute:
Activates when uninitialized data is used in computations. This avoids trapping on uninitialized memory copies, but traps when an ALU instruction is used on the data.

branch:
Activates when a branch occurs on a flag state that is undefined due to being based on uninitialized data.

ea (effective address):
Activates when an indexed addressing mode uses an index register with a value based on uninitialized data.

hwstore:
Activates when a store to a hardware register occurs with a value based on uninitialized data.

Traps can be enabled early or at default time. The default enabled state (on) tracks memory state from reset but only activates traps once the OS has initialized or playfield DMA is enabled, whichever comes first. This avoids false positives on problematic OS code, notably the RAM size test. Alternatively, traps can be enabled early, which enables traps immediately after reset. More complex conditions can be enabled by issuing the hmt command off of breakpoints.

The heat map engine tracks uninitialized state in registers, flags, and memory on a bit basis. For instance, LSR on an uninitialized location will result in the C flag and bits 0-6 tagged as uninited, but bit 7 and the N flag will be marked valid. Validity state can also be tracked from one memory location to another through registers during a memory copy. However, the engine does not track the memory map state and can be confused by additional hardware, high level emulation, or cartridge ROMs.

The heat map must be enabled for heat map traps to work. Since the heat map traps the source of values, it should be enabled from boot in order to trace all data flows correctly. The hmp and hmu commands can be used to force the validity state of memory to correct mistakes.

See also: hme (heat map enable), hmp (preset heat map range), hmu (unset heat map range), hmr (show heat map register state)

hmu Unset heat map range

Marks a range of memory in the heat map as invalid.

```
hmu <xaddr> L<length>
```

See also: hmp (preset heat map range)

ib Input byte

Read from a memory address in CPU address space with side effects.

```
ib <address>
```

Reads from a memory address using normal CPU core read rules, activating any side effects normally associated with reads from that location. Normally, reads in the debugger have side effects disabled so that they do not change simulation state; the 'ib' command allows those side effects to occur as usual. This includes clearing PIA interrupts when PORTA/B is read and switching cartridge banks on reads from the CCTL region with certain cartridge types.

See also: db (display bytes)

k Show call stack

Display a list of return addresses on the stack.

```
k
```

The call stack is computed by virtualized execution, so it may be incorrect if bankswitching or other complex techniques are involved. Note that because return addresses are displayed, the call stack will not show parent call sites to children that never return in normal execution.

lfd, lfe, lfl, lft Logging filter control

```
Display or modify logging filter settings.

lfd <channel>|* Disable logging channel
lfe <channel>|* Enable logging channel without timestamps
lfl List logging channels

The logging filter controls which events from the simulation are shown
in the debug output window during execution.

lft [-t] [-c] [-u] [-r] <channel>|*

Enables a logging channel with prefixes indicating running status:

-t Include beam position timestamp.
-c Include cassette tape position.
-u Include timestamp in microseconds (us).
-r Include raw 32-bit timestamp in cycles.

If no switches are supplied, -t is assumed.
```

lm List modules

```
Display a list of currently known modules and any loaded symbols.

lm
```

ln List nearest symbol

```
Display the closest symbol to a given address.

ln <address>

The closest symbol is the one with the highest address that is equal to
or lower than the given address.
```

m Move memory

```
Move a block of memory to another location.

m <address> L<length> <dest-address>

Memory in the source block is copied to a new block starting at the
destination address. If the source and destination blocks overlap, a
descending copy is used if needed to copy correctly. This is not
guaranteed if aliased memory windows are used, such as copying from
$4000 to x:$100 with the CPU extended memory window active.
```

o Step over

Execute one instruction, stepping over any subroutine calls.

```
o
```

This is the same as (t)race, except that if a JSR or an interrupt is encountered, execution continues until the subroutine exits.

r Registers

Display or modify register values.

```
r (Display registers)
r <reg> <value> (Modify register)
```

For 6502-derived CPUs, the registers that can be modified are: PC, A, X, Y, S, and P. The flags in the P register can also be individually toggled:

p.n Negative (sign) flag
p.v Overflow flag
p.d Decimal mode flag
p.i Interrupt mask flag
p.z Zero flag
p.c Carry flag

The break flag cannot be modified as it does not exist in the P register,
only in the image of P pushed on the stack during interrupt entry.

In 65C816 mode, the following additional registers are available:

c 16-bit accumulator (B:A)
d Direct page register
b/dbr Data bank register
k/pbr Program bank register
p.m Accumulator size flag
p.x Index size flag
e Emulation flag

For Z80 coprocessors, the available registers are: A, F, B, C, D, E, H, L,
I, R, AF, BC, DE, HL, IX, IY, SP, A', B', C', D', E', H', L', AF', BC', DE',
and HL'.

For 8048 coprocessors, the available registers are: A, PSW, R0-R7.

s, sa, si, sw — Search memory

Searches memory for a specific pattern.

```
s <xaddress> L<length> <value>... (search bytes)
sa <xaddress> L<length> "string" (search ATASCII)
si <xaddress> L<length> "string" (search INTERNAL)
sw <xaddress> L<length> <value>... (search words)
```

All starting addresses of blocks of memory that contain the given byte, word, or string pattern.

For s (search bytes), expressions must evaluate to unsigned byte range (0-255); use the low byte (<) operator if necessary to forcibly truncate values.

Examples:

```
s 0 L10000 00 04 A5
s E800 L2800 <vimirq >vimirq
sw E800 L2800 vimirq
sa 0 L10000 "BOOT"
```

st — Static trace

Statically traces program execution and marks labels.

```
st [-m] <baseaddr> [<restrictbase> L<restrictlength>]
```

The st command examines program code and finds all static program traces by following absolute JMP/JSR and relative branch instructions. The custom symbol table is then populated with labels for each jump or branch target.

If -m is specified, a label is also added for the initial base address.

Static tracing is only capable of following traces by references that are statically embedded within the code. Dynamic references through jump tables or vectors cannot be seen by the static tracer.

If restrictbase and restrictlength are specified, only traces within the given restricted range are followed. This is useful for contraining tracing to within a specific module and avoiding bogus tracing into other regions, such as RAM where dynamic code is kept.

stp Static trace PBI

Statically traces the current parallel bus interface (PBI) firmware.

```
stp
```

The stp command is equivalent to running the static trace (st) command as follows:

- Address range is restricted to D800-DFFF.
- The SIO and IRQ vectors at $D805 and $D809 are traced.
- The init vector at $D819 is traced.
- The CIO vectors at $D80D-D818 are traced (with +1 adjustment).

The PBI firmware must be selected and visible when the stp command is executed.

See also: st (static trace)

t Trace (step one instruction) (F11)

Execute one instruction, stepping into a subroutine or interrupt.

```
t
```

u Unassemble

Disassemble CPU code at a given address.

```
u [-e|[-m8|-m16][-x8|-x16]] [-p] [-n] [-m <mode>]
[<address> [L<length>]]
```

If no address is given, disassembly continues from the end of the last continuable command.

In 65C816 mode, instruction encoding depends upon the current CPU mode. The disassembler attempts to track the current M/X/E flags, but can easily
be misled. The following switches can be supplied to guide the decoder:

-m8 Assume E=0/M=0 (native mode, 8-bit memory/accumulator)
-m16 Assume E=0/M=1 (native mode, 16-bit memory/accumulator)
-x8 Assume E=0/X=0 (native mode, 8-bit index registers)
-x16 Assume E=0/X=1 (native mode, 16-bit index registers)
-e Assume E=1 (emulation mode)

By default, the disassembler detects common REP/SEP patterns to predict the M/X mode. This automatic prediction can be disabled with -p.

-n disables label decoding for the PC address and operands.

-m changes the disassembly mode to a different CPU. Currently supported modes are: 6502, 65c02, 65c816, z80, 8048, and 6809.

vta, vtc, vtl, vtr — Verifier target control

Add, clear, or list verifier allowed OS entry targets.

```
vta <address> (Add allowed target)
vtc <address> (Clear allowed target)
vtc * (Clear all allowed targets)
vtl (List allowed targets)
vtr (Reset allowed targets)
```

Manages the list of allowed kernel entry targets allowed by the verifier.
The verifier checks for improper dependencies on internal OS routines by forcing a debug break if it sees a control transfer into the OS ROM with an entry point not on the allowed target list. This detection may misfire with an OS ROM that contains extra entry points or with a routine that self-modifies its code with a vector address. The vt* commands allow the target list to be modified to exclude valid targets from detection.

wb, wc, wl, ww — Watch data

Continuously display the value of a memory location or expression.

```
wb <address> (Watch byte)
ww <address> (Watch word)
wl (List watches)
wc <index>|* (Clear watches)
```

The wb and ww commands display the contents of a byte or word at the given address. Watched values are sampled and displayed each frame. Up to eight watches can be active at any one time.

See also: ? (evaluate expression), wx (watch expression)

wx — Watch expression

Continuously display the value of an expression.

```
wx [-x8|x16|x32] <expr>
```

The wx command displays the value of an expression, which is re-evaluated every frame. The expression can include memory reads (db/dw/etc.), which is useful for reading variables too complex for wb or ww.

By default, the computed value is displayed in decimal. The -x8, -x16, and -x32 options instead format the computed value as an 8-bit, 16-bit, or 32-bit hex value.

See also: wb (watch byte), ww (watch word), wl (list watches), wc (clear watches)

x — Examine symbols

List symbols whose names match a given pattern.

```
x <pattern> (Examine symbols with name pattern)
x <pat>!<pat> (Examine symbols with module and name patterns)
```

The x command lists symbols according to a wildcard pattern, with ? and * characters representing one and zero+ unknown characters. If a simple pattern is supplied, all symbols matching that pattern are reported; if a module pattern is supplied, only symbols matching both the module and name patterns are reported.

Examples:

```
x * Lists all symbols
x hardware!* Lists all hardware symbols
x kernel*!a* Lists all symbols starting with "a" in kernel modules
```

See also: .loadsym, .unloadsym, lm (list modules)

ya, yc, yd, yr, yw — Manage manual symbol table

Add or remove symbols from the manual symbol table.

```
ya <name> <address> [L<length>] (Add manual symbol)
yc (Clear manual symbols)
yd <address> (Delete manual symbol)
yr <filename> (Read manual symbol table)
yw <filename> (Write manual symbol table)
```

The manual symbol table allows on-the-fly naming of addresses in memory. Any symbols added to the manual symbol table can be used for address specification and decoding as any other symbol.

.antic — Display ANTIC status

.bank — Show memory bank state

.base — Set numeric parsing base

Selects decimal or hexadecimal as the default parsing base for numbers.

```
.base dec / 10 (Set base 10 for numbers)
.base hex / 16 (Set base 16 for numbers)
.base mixed (Set base 10 with 16 shortcut for numbers)
```

The .base command changes whether numbers in expressions are by default parsed as decimal (base 10) or hexadecimal (base 16). The default is mixed, which selects hex for simple numbers/addresses and decimal for complex expressions. This permits unprefixed hex for simple addresses while avoiding unexpected hex values in expressions.

Examples of how various expressions are interpreted:

Expression	Hex	Dec	Mixed
10	$10	$0A	$0A
$10	$10	$10	$10
A5	$A5	error	$A5
(10)	$10	$0A	$0A
(A5)	$A5	error	error
01:2000	$012000	$0107D0	$012000
$01:2000	$012000	$012000	$012000

See also: ? (evaluate expression)

.basic — Dump BASIC table pointers

Displays the addresses and sizes of BASIC tables.

```
.basic
```

See also: .basic_dumpline, .basic_vars, .basic_dumpstack, .basic_rebuildvnt, .basic_rebuildvvt, .basic_save

.basic_dumpline Dump BASIC program line

Lists the statements contained within a BASIC line.

```
.basic_dumpline [<address> | * | 0] [-o <offset>] [-c] [-t] [-k]
```

The optional address specifies the beginning of the line, including the line number. If no address is specified, the continuation address from the last command is used.

Using * for the address uses the current line address (STMCUR). Zero specifies the beginning of the program (STMTAB).

-o specifies an optional byte offset from the beginning of the line. Any token that begins that that byte offset is marked in the output with >>.

-c gives continuous output until the end of the program.

Tokens from Atari BASIC, BASIC XL, and BASIC XE are supported by default.
-t specifies decoding of TurboBasic XL tokens instead of Basic XL/XE tokens.

-k displays a hex dump beside each token.

See also: .basic, .basic_vars

.basic_dumpstack Dump BASIC runtime stack

Lists pending GOSUB and FOR..NEXT loops on the BASIC stack.

```
.basic_dumpstack [-a] [-t]
```

-a decodes line references as addresses instead of line numbers. Altirra BASIC and Turbo-Basic XL use addresses on the runtime stack during execution.

-t decodes stack frames using Turbo-Basic XL format. This adds support for REPEAT, WHILE, DO, and EXEC frames, as well as expanded FOR frames with 256 variable support.

See also: .basic

.basic_rebuildvnt — Rebuild BASIC variable name table

Reinitializes the BASIC variable name table (VNT) based on the variable value table (VVT).

```
.basic_rebuildvnt [-t]
```

Deletes all existing variable names and replaces them with new variable names, based on the variable type information in the VVT. This can repair a program with a damaged VNT, at the cost of losing any existing variable names. The statement table, array table, and runtime stack are relocated in the process.

The type byte in the VVT must be valid for this command to work. You can't rebuild both the VVT and the VNT if both are trashed.

Using this command on a running BASIC program is likely to produce fireworks and is not recommended.

-t enables support for TurboBasic XL labels.

See also: .basic_rebuildvvt

.basic_rebuildvvt — Rebuild BASIC variable value table

Reinitializes the BASIC variable value table (VVT) based on the variable name table (VNT).

```
.basic_rebuildvvt
```

Most of the data in the VVT is ignored after LOAD, but the variable type and index need to be correct. If they are not, BASIC can act erratically.

The .basic_rebuildvvt command resets the mode and index bytes on each variable entry based on the order of the names in the VNT and the type characters on each entry, i.e. whether they end in $ or (. The VVT can be rebuilt from scratch even if it is completely corrupted as long as the VNT is intact.

The names in the VNT must be valid for this command to work. You can't rebuild both the VVT and the VNT if both are trashed.

See also: .basic_rebuildvnt

.basic_save Save BASIC program

```
.basic_save <path>
```

Saves a BASIC program to a file on the host (not on the emulated disk). The file produced is the same format produced by Atari BASIC.

A check is made during the save process for an abnormally large argument stack region, which is caused by Atari BASIC rev. B when saving.
If detected, the argument stack region is corrected back to its normal size of 256 bytes.

This command cannot be used with Basic XE when Extend mode is enabled.

.basic_vars Dump BASIC variables

Dumps the contents of the Atari BASIC variable name table (VNT).

```
.basic_vars
```

The token and name of each variable is displayed.

.batch Run debugger batch script

Runs the contents of a text file as a series of debugger commands.

```
.batch <filename>
```

.beam Show ANTIC scan position

.ciodevs Dump Central Input/Output (CIO) device list

Dumps the list of CIO devices in the HATABS database.

```
.ciodevs
```

.covox Dump Covox sound extension status

.crc Compute CRC of memory range

Computes Cyclic Redundancy Check (CRC) values for data in a memory range.

```
.crc [-i initial_value] address L<length>
```

The CRC-16-CCITT and CRC-32 values are computed and reported. These particular CRC algorithms are used by Atari physical disk formats and for identifying ROM images, respectively. Both CRCs are computed conventionally, as a modulus of the message appended with zero bytes at the end where the CRC would be located. The CRC-32 is additionally inverted at the end.

If -i is specified, its argument is used as the initial value for the CRC computation instead of -1.

See also: .sum

.ctc Dump Z8430 CTC status

```
.ctc
```

Displays the status of any Z8430 Counter/Timer Circuit (CTC) chips in the system.

.diskdumpsec Dump floppy disk sector data

Displays raw sector data.

```
.diskdumpsec [-d <drive>] [-i] [-I] <virt-sector>
```

Reads a raw sector from disk by virtual sector number and displays its contents. If phantom sectors are present, the first one is used. Dumping a sector does not disturb the drive's timing state.

If -i is specified, the data is inverted before being displayed. This is useful for disk formats that are non-inverted on disk and thus inverted from the view of the computer (Atari 815, Indus GT CP/M).

-I displays characters as INTERNAL instead of ATASCII. Bit 7 is also ignored.

WARNING: While this command doesn't disturb timing state, it will change virtual disk image state when reading sectors off a drive with a mounted folder.

See also: .diskorder (set forced phantom sector ordering), .disktrack (show sector order within track)

.diskorder Set forced phantom sector ordering

Overrides default phantom sector ordering with a predefined order.

```
.diskorder <sector> (Restore default sector ordering)
.diskorder <sector> <indices...> (Override sector ordering)
```

By default, phantom sectors within a disk track are returned either by their storage order in the disk image (.pro format) or according to sector timing (.atx format). The .diskorder command allows the ordering to be overridden in order to diagnose or correct load failures due to incorrect sector order.

Phantom sector indices start with 1 and correspond to the Nth physical sector within the disk image. This command causes virtual sector 43 to load its four phantom sectors in stored order:

```
.diskorder 43 1 2 3 4
```

This command reverses the load order:

```
.diskorder 43 4 3 2 1
```

The DISK logging channel is useful for examining the sector load order initiated by a program.

See also: lfe (enable log channel), lft (enable tagged log channel), .disktrack (show sector order within track)

.diskreadsec Read sector from floppy disk

Reads a sector from floppy disk into memory.

```
.diskreadsec [-d <drive>] <virt-sector> <address>
```

Reads a raw sector from disk by virtual sector number and writes it into memory. If phantom sectors are present, the first one is used. Dumping a sector does not disturb the drive's timing state.

For sectors that have different virtual and physical sector sizes, such as boot sectors, the virtual sector size is used. That is, on a double density disk, the read size is 128 bytes for sectors 1-3 and 256 bytes for all other sectors.

WARNING: While this command doesn't disturb timing state, it will change virtual disk image state when reading sectors off a drive with a mounted folder.

See also: .diskwritesec (write sector to floppy disk)

.disktrack Show sector order within track

Displays all sectors within a track on D1: in rotational order.

```
.disktrack [-d <drive>] <track>
```

The sectors are shown with their virtual sector number and phantom index, followed by the position of the sector in fractions of a rotation. For a standard drive at 288 RPM, there are about 4.8 rotations per second.

See also: .diskorder (set phantom sector ordering),

.diskwritesec Write sector to floppy disk

Writes a sector from memory to a floppy disk.

```
.diskwritesec [-d <drive>] <virt-sector> <address>
```

Writes memory to a raw sector from disk by virtual sector number. If phantom sectors are present, the first one is used.

See also: .diskwritesec (write sector to floppy disk)

.dlhistory Show ANTIC display list execution history

Displays history of display list instructions executed by ANTIC.

```
.dlhistory
```

Unlike .dumpdlist, .dlhistory shows past display list history even if the display list has been modified in memory. It also shows some recorded information not found in the display list itself, such as the HSCROL, VSCROL, and DMACTL states when the display list instructions were executed.

In the event that a multi-line jump instruction occurs, the jumps on each scanline are displayed:

```
120: BC30 BDF8 0 4 22 A2
124: BC31 BE20 0 4 22 81
125: BC34 ---- - - 22 81
126: 0202 ---- - - 22 81
127: C0CD ---- - - 22 81
128: 4068 ---- - - 22 81
129: 0000 BE20 0 4 22 00
```

.ds1305 Show DS1305 real-time clock status

Displays status of the DS1305 real-time clock chip in the SIDE or Ultimate1MB hardware.

```
.ds1305
```

.dma Show current ANTIC DMA pattern

Displays the pattern of DMA cycles performed by ANTIC for the current scan line.

```
.dma
```

The DMA pattern can change in the middle of the scan line depending on when certain critical cycles are passed and if any pertinent registers are rewritten. For instance, a mid-line write to DMACTL that changes the playfield width can change the DMA pattern.

See also: .dmamap (show ANTIC DMA activity map)

.dmabuf Show ANTIC DMA line buffer

Displays the contents of the internal line buffer within ANTIC.

```
.dmabuf
```

See also: .dma (show current ANTIC DMA pattern)

.dmamap Show ANTIC DMA activity map

Displays the pattern of DMA cycles performed by ANTIC for the current frame.

```
.dmamap
```

ANTIC DMA analysis must be enabled in the View menu for this command to work, as otherwise DMA cycle patterns are not captured.

Each line is of the form:

```
8: .*......................~ ~..................... | 10:104
```

The numbers at the end are DMA and non-DMA cycle counts, respectively.

Unlike the .dma command, the .dmamap command always shows the exact DMA cycle pattern that occurred, complete with any mid-scanline changes. However, it cannot show upcoming DMA cycles like the .dma command can.

See also: .dma (show current ANTIC DMA pattern)

.dumpdlist Dump ANTIC display list

Disassemble an ANTIC display list stored in memory.

```
.dumpdlist [-n] [<address>]
```

Options:

-n Do not collapse groups of identical mode lines.

If no address is supplied, the current ANTIC display list pointer is used.

.dumpdsm Dump disassembly to file

Disassemble a range of memory as CPU instructions to a file.

```
.dumpdsm [-c] [-l] [-p] [-n] [-s] [-t] <filename> <addr> L<length>
```

The following options can be specified:

-c Include code bytes.
-l Use lowercase opcode names.
-n Do not decode labels.
-p Include PC address.
-s Separate routines with blank lines after jumps and returns.
-t Use 4 character tabs.

.dumpsnap Create bootable snapshot image

Create a bootable snapshot image from the current simulator state.

```
.dumpsnap [-u] <path.atr>
```

Options:

-u Disable compression and use uncompressed blocks

Records hardware and memory state to a disk image with a loader that restores the state on boot. The snapshot image contains the base 64K of memory and requires a 128K system to boot (the extra memory is used by the loader). The loader attempts to reconfigure the hardware to match the snapshot state and then resume the running program.

Not all state is restored precisely and load success depends on the exact state and program code. For best results, the same OS ROM should be used, no cartridge should be present, and the snapshot should be taken at the beginning of the NMI handler for the vertical blank interrupt.

.echo Display message to console

Writes a string to the console window.

```
.echo <strings...>
```

Quoted strings are displayed without quotes. To display a string with a quotation mark in it ("), use escaped string syntax. Escaped strings are quoted strings that are prefixed with \ and have \n, \", or \\ escapes inside them.

Example:

```
.echo foo prints: foo
.echo foo bar prints: foo bar
.echo "foo bar" prints: foo bar
.echo \"\"foo bar\"" prints: "foo bar"
.echo \"x\ny" prints: x
y
```

See also: .printf

.fpaccel Control floating-point math pack acceleration

Selectively enables or disables math pack acceleration routines.

```
.fpaccel (List enable/disable status)
.fpaccel -e address (Enable acceleration for specific routine)
.fpaccel -d address (Disable acceleration for specific routine)
```

Enables or disables specific routine hooks when floating-point acceleration is enabled. The hook is selected by the calling address of the routine. By default, all supported routines are enabled for acceleration. These hooks and the .fpaccel command only have an effect when the global option for FP acceleration is enabled.

Note that in mixed radix mode, FADD is the address $FADD; (FADD) must be used to select the symbol.

.gtia Display GTIA status

.ide Display IDE emulator status

Displays the current state of the emulated IDE device.

```
.ide
```

.ide_dumpsec Dump IDE raw sector

```
Dumps the contents of a raw IDE hard disk sector.

.ide_dumpsec [-l] <lba>

Reads a sector by linear block address (LBA) and displays the contents.
This bypasses the ATA interface, so it can be done unobtrusively even if
the emulated device is in reset state or in the middle of a command.

-l Dump only every other byte (16-bit mode on 8-bit bus).

See also: .ide_rdsec, .ide_wrsec
```

.ide_rdsec Read IDE sector into memory

```
Reads an IDE sector into CPU memory.

.ide_rdsec [-l] <lba> <address>

Reads a sector by linear block address (LBA) into CPU memory. This
bypasses the ATA interface, so it can be done unobtrusively even if the
emulated device is in reset state or in the middle of a command. 512
bytes are read by default, or 256 if -l is specified.

-l Read only every other byte (16-bit mode on 8-bit bus).

See also: .ide_dumpsec, .ide_wrsec
```

.ide_wrsec Write IDE sector from memory

```
Writes an IDE sector from CPU memory.

.ide_wrsec [-l] <lba> <address>

Writes a sector by linear block address (LBA) from CPU memory. This
bypasses the ATA interface, so it can be done unobtrusively even if the
emulated device is in reset state or in the middle of a command. 512
bytes are read by default, or 256 if -l is specified.

-l Write only every other byte (16-bit mode on 8-bit bus). The
high byte is set to $FF.

See also: .ide_dumpsec, .ide_wrsec
```

.iocb Display CIO I/O control blocks

Displays the contents of the nine IOCBs used by CIO.

```
.iocb
```

The zero-page IOCB, or ZIOCB, displayed as ZP. It is only valid during CIO operation.

A device name including an ID and a tilde, e.g. $50~R:, is reported for a provisionally open IOCB. An IOCB is provisionally opened when the OS successfully receives a device response to a type 4 poll for a missing CIO device on the SIO bus. Further I/O to this IOCB with the HNDLOD variable set will trigger a load of the handler from the device into memory. The reported ID is the SIO address of the device to be used for handler loading.

See also: .tracecio, .ciodevs

.kmkjzide Display KMK/JZ IDE / IDEPlus 2.0 status

Displays the status of the KMK/JZ IDE or IDEPlus 2.0 device.

```
.kmkjzide
```

.loadksym Load kernel symbols

Loads kernel symbols for the original 10K OS.

```
.loadksym <file>
```

Loads a symbol file for a module at $D800-FFFF.

.loadobj Load executable object

Loads an executable object from a file.

```
.loadobj <file>
```

<file> must be an executable object in Atari DOS executable format. INITAD and RUNAD segments are skipped and no changes to execution state are made other than to load program segments into memory. This is intended to allow helper modules to be loaded into RAM.

See also: .readmem, .writemem

.loadstate Load simulation state

```
.loadstate <path>
```

Loads emulation state from the given save state file. This is the same format as used when the Load State command in the UI is used, with the default extension being .atstate2.

The .loadstate command cannot load the old Altirra V1 save state format (*.altstate). Use the UI Load State command to load old save state files.

See also: .savestate

.loadsym Load module symbols

Loads module symbols from a symbol file.

```
.loadsym <file>
```

Two types of symbols are supported. A labels file allows the debugger to match addresses to labels, while a listing file permits source-level debugging. Both label and listing files can be loaded at the same time, and multiple sets of symbols can also be loaded.

.logopen Open log file

Records all console output to a log file.

```
.logopen <file>
```

Closes any existing log file and opens a new one. All subsequent console output is recorded to the file until the log file is closed.

See also: .logclose

.logclose Close log file

Closes any currently open log file.

```
.logclose
```

See also: .logopen

.map Show memory map layers

```
Displays memory layers active in the memory mapper

.map

This displays the memory layers that are being tracked internally in
the emulator.
```

.netpcap, .netpcapclose Capture packet trace from emulation network

```
.netpcap <file> (Begin packet trace to file)
.netpcapclose (End packet trace)

Begins capturing all packets from the emulation network and logging them
to a packet trace file. The file is written in libpcap-compatible
format,
which can be used with tools such as tcpdump and Wireshark.
```

.netstat Display network connection status

```
Shows UDP/IP and TCP/IP connections on the emulated network.

.netstat

Displays a list of connections maintained by the gateway on the emulated
network. This only applies if DragonCart emulation is active.

Five columns are shown:

Proto:
Protocol for connection (TCP/UDP).

Local address:
Emulation-side address as seen by the gateway. This is typically
the address provided by the program driving the DragonCart and
will be on the emulation subnet.

Foreign address:
The other address of connection, either the gateway itself or an
external host.

State:
TCP connection state

NAT address:
Host address used by NAT gateway for external connections. This
is the address that the external server sees the connection from.
```

.onexeclear, .onexelist, .onexeload, .onexerun

Executable command triggers

```
Queue commands for when an executable loads or runs.

.onexeclear (Clear queued commands)
.onexelist (List queued commands)
.onexeload <command> (Queue command prior to executable load)
.onexerun <command> (Queue command prior to executable run)

This allows debugger commands to be automatically issued when
an executable loads or runs. The on-load commands are issued
before the first segment is loaded, while the on-run commands
are issued before the run vector is activated. Commands are
removed from the queue after they are run.

The on-exe commands only trigger when an executable is loaded
by the simulator itself. They do not activate when an executable
is loaded by DOS, which is invisible to the simulator.
```

.pagesums Display checksum map of memory pages

```
Display a map of checksum bytes for all 256 pages of memory.

.pagesums

Each byte in the map is a checksum of all bytes in that page. This
allows
for quick verification against another instance to see which pages are
the same or different. At the end of each line is a row checksum, which
is
computed over all of the page checksums in that row; this allows for
quick
checking of rows (16 x 256 bytes = 4KB).

The checksums are computed over the CPU memory space, so they will show
whatever data is currently visible to the CPU at that address in the
current
cycle.

For convenience, a zeroed page has a checksum of $00.
```

.pathdump, .pathrecord, .pathreset, .pathbreak

Manage execution path recording

```
Record and dump instruction paths executed by the CPU.

.pathrecord [on|off] (Show or change path recording setting)
.pathreset (Clear recorded paths)
.pathdump <file> (Dump path disassembly to a file)
.pathbreak [on|off] (Show or change new path break setting)

Path recording, when enabled, marks the addresses of branch targets
and subroutines during execution, making it easier to follow
execution flow in a disassembly and identifying which memory areas
are confirmed to contain code. When enabled, the diassembly will
also show pseudo-labels for any addresses not already marked with
a symbol.

The .pathbreak command permits halting execution whenever a new
path is encountered. This is handy for identifying the exit path
in a large frame loop, as the body of the loop can be captured and
then .pathbreak enabled to capture the exit path.
```

.pbi Display Parallel Bus Interface (PBI) status

.pclink Display PCLink status

.pia Display Peripheral Interface Adapter (PIA) status

.pokey Display POKEY status

.printf Display message with formatted fields

Writes a string to the console window with formatted fields.

```
.printf "format-string" [field...]
```

Format-string can contain formatted fields like the C library function printf(). Formatting specifiers are of the form:

```
%[flags][width][.precision][length]type
```

Flags:

```
0 Apply zero padding to right-justified string
# Prefix %x output with 0x and %X output with 0X
+ Always display +/- sign on numbers
<space> Prefix positive numbers with a space
- Left-justify formatted field
```

Width is a positive number specifying the minimum characters to display for the field. If the formatted field is shorter than the width, the field is padded according to flags, with right justification with spaces being the default.

Precision is a non-negative number specifying minimum number of digits to display for numeric types. Zeroes are prepended to the number if it has fewer digits than the specified precision.

Both width and precision may be specified as variable (*) in order to set them via separate expressions. If variable, the width and precision arguments come before the value argument, in that order. The values are clamped to avoid disasters.

Length specifies the size of an integer field:

```
hh Byte (8-bit)
h Word (16-bit)
l Long (32-bit) (default)
```

Type specifies the field type:

```
%b Binary number
%c ASCII character (only 20-7E; others produce '.')
%d Signed decimal
%e Decimal float by address, exponential notation
%f Decimal float by address, floating-point notation
%g Decimal float by address, general notation
%i Signed decimal (equivalent to %d)
%s ATASCII string buffer by address ($20-7F)
%S ATASCII string buffer by address (high byte term.)
%u Unsigned decimal
%x Hexadecimal (lowercase)
%X Hexadecimal (uppercase)
%y Symbol address
```

Unlike the C function printf(), .printf always prints a newline after the formatted string. Also, note that %e/%f/%g require the address of a decimal float and not the value of a binary float.

Example:

```
.printf "Display list vector: $%04x" dw(vdslst)
```

See also: .echo

.profile_beginframe, .profile_endframe

Trigger profiler frames

Starts or ends a frame in the CPU profiler.

```
.profile_beginframe Trigger start of profiler frame
.profile_endframe Trigger end of profiler frame
```

The .profile_beginframe and .profile_endframe commands can be used with conditional breakpoints to trigger frame boundaries with complex conditions. Note that any frame triggers set in the profiler itself will still fire even when these commands are used. If profiling is not active, these commands are silently ignored.

.rapidus Display Rapidus status

Displays status of the Rapidus Accelerator device, if present.

```
.rapidus
```

.readmem Read memory from disk

Read a block of data from a file on disk into memory.

```
.readmem <path> <xaddress> [L<length>]
```

The address may use extended memory syntax, i.e. v:4000 for VBXE memory.

See also: .loadobj

.reload Reload symbol files

Re-reads symbol files for all currently loaded symbols.

```
.reload
```

See also: lm (list modules), .loadsym, .unloadsym

.restart Restart emulated system

```
Performs a cold reset.

.restart
```

.riot Dump 6532 RIOT status

```
.riot

Displays the status of any 6532 RAM Input/Output Timer (RIOT) chips
in the system, particularly those in disk drives (full drive emulation
only).
```

.savestate Save simulation state

```
.savestate <path>

Saves emulation state to the given save state file. This is the same
format
as used when the Save State command in the UI is used, with the default
extension being .atstate2.

Unlike the UI, the .savestate command does not attempt to step the
simulation to the next CPU instruction boundary and will save a state
mid-instruction.

See also: .loadstate
```

.side3 Dump SIDE3 status

```
.side3

Displays internal SIDE3 device state.
```

.sdx_loadsyms Load SpartaDOS X symbols

```
Loads symbols from the SpartaDOS X symbol table.

.sdx_loadsyms [address]

By default, this follows the symbol chain starting with the pointer
at address DOSVEC+$127. An alternate override address for the pointer
can be specified. The loaded symbols are placed in a module called
SDX. If this module exists, it is cleared before being reloaded.

NOTE: This command can only see the full set of symbols if the SDX
library is banked in. Otherwise, only resident symbols outside of
the library bank will be seen. This isn't a problem during normal
operation because the library bank has to be enabled for either
symbol resolution or to call library functions.
```

.sio Dump SIO device control block (DCB)

Displays the contents of the SIO device control block.

```
.sio [-b]
```

The -b option gives one-line abbreviated output.

.sourcemode Switch between source and disassembly level debugging

Selects or displays the current source debugging mode.

```
.sourcemode [on|off]
```

The debugger can optionally jump to source code whenever appropriate symbols are available. By default, this happens only when single stepping or resuming execution from a source code window. By using the .sourcemode command and the -n option of the g (Go) command, however, you can force source mode debugging. This is most useful in scripts.

See also: g (Go)

.sprintf Construct message with formatted fields

Constructs a string with formatted fields.

```
.sprintf "format-string" [field...]
```

Expands format tokens in the format-string with the supplied fields and stores it in the @ts variable. This variable can then be used by subsequent commands that take a text string, such as .printf, .sprintf, and bt.

The format-string is the same format as that used by the .printf command.

See also: .printf, bt

.sum Compute sum of memory range

Computes the sum of bytes in a memory range.

```
.sum [-w] address L<length>
```

All bytes within the memory range are added and both the binary sum and the one's complement (carry wraparound) sum are reported.

If -w is specified, a sum of words is computed instead. The inverted and byte-swapped checksum is also reported for TCP/IP purposes.

.tape Display cassette tape deck status

.tapedata Display cassette tape data

Displays data from the currently loaded cassette tape.

```
.tapedata [-t] [-b <baud> [-d]] [-r <pos>] [-p <pos>] [-s <pos>] [-y]
```

```
-t Display bit transitions
-b <baud> Decode data at given baud rate
-d Display only decoded bytes
-r <pos> Decode at offset in milliseconds from current position
-p <pos> Decode at absolute position in milliseconds
-s <pos> Decode at absolute position in samples
-y Bypass FSK decoder (turbo mode)
```

The .tapedata command displays the data near the current tape position. By default, the raw internally stored data bits are displayed at 4Kbaud. If the -t flag is specified, only the locations of transitions are displayed.

The -b flag specifies byte decoding at a specified baud rate and causes the debugger to attempt to identify byte locations through start and stop bits. The -d flag causes only decoded bytes and not intermediate data bits to be displayed.

The decoding algorithm is similar but not the same as that which actually occurs during simulation, which has an additional low pass filter on the serial input based on the serial rate. The debugger can also misdecode the initial bytes due to starting from ground state.

See also: .tape

.tracecio Toggle CIO call tracing

Logs calls to the operating system's Central Input/Output (CIO) facility.

```
.tracecio [on|off]
```

If enabled, calls to the CIOV vector are trapped and the parameters for the active IOCB are reported to the console.

See also: .tracesio

.traceser Toggle POKEY serial I/O tracing

Enables or disables POKEY serial I/O (SIO) tracing.

```
.traceser [on|off]
```

This command is deprecated; the SIODATA logging channel is recommended instead.

See also: lfe (log filter enable)

.tracesio Toggle SIO call tracing

Logs calls to the operating system's Serial Input/Output (SIO) facility.

```
.tracesio [on|off]
```

If enabled, calls to the SIOV vector are trapped and the parameters in the device control block (DCB) are reported to the console.

See also: .tracecio

.ultimate Dump Ultimate1MB status

Displays the state of the Ultimate1MB expansion hardware.

```
.ultimate1mb
```

See also: .ds1305

.unloadsym Unload module symbols

Unloads symbols for a module from the debugger.

```
.unloadsym <module-name>|<module-id>
```

See also: lm (list modules)

.vbxe Display VBXE status

.vbxe_bl Display VBXE blit list (BL)

```
Displays the current or an alternate VBXE blit list.

.vbxe_bl [-c] [<address>]

-c Use compact output format

The address must be a VBXE address (v:offset) or a CPU address currently
within a CPU-enabled MEMAC window. If the address is omitted, the
starting address in the VBXE_BL_ADR0-2 registers is used.
```

.vbxe_pal Display VBXE palette

```
Displays entries from the VBXE palette.

.vbxe_pal [<offset>] [L<length>]

Displays RGB colors from the VBXE palette memory, which has four banks
of 256 colors with 21 bits each. The values are displayed in RGB form as
24-bit integers; the LSB of each byte is zero as it does not exist.

By default, all 256 colors are displayed from the bank specified by
the PSEL register. If offset is specified, it is a value from 0-1023
selecting a specific color and bank ($304 = color $04 from bank 3).
Length
specifies the number of colors to display.
```

.vbxe_traceblits Toggle VBXE blit tracing

```
Logs each time VBXE starts a blit.

.vbxe_traceblits off|on|compact
```

.vbxe_xdl Display VBXE extended display list (XDL)

```
Dumps the contents of the current VBXE extended display list.

.vbxe_xdl

The XDL address is obtained from the current VBXE_XDLn registers.
```

.vbxe_xdlhistory Display VBXE extended display list (XDL) history

```
Displays the execution history of the VBXE XDL up to the current point.

.vbxe_xdlhistory
```

.vectors — Display kernel vectors

.warmreset — Warm reset simulation

```
Resets the simulation hardware (XL/XE) or triggers the System Reset
button logic (400/800).

.warmreset

See also: .restart
```

.writemem — Write memory to disk

```
Write a block of memory to a file on disk.

.writemem <path> <xaddress> L<length>

The address may use extended memory syntax, i.e. v:4000 for VBXE
memory.
```

.help — Display help in debugger

```
List the commands available in the debugger.
```