altairsim — The Monitor

2026-10-24 · 6f09484

A simulation of the MITS Altair 8800 and the S-100 bus

The monitor 1
Commands resolve by prefix 1
Editing the command line 2
Repeating the last command: `.` 4
Reaching the host: `!` 4
Numbers: one fixed rule 4
Naming a board: `<id>[:<unit>]` 6
Seeing the machine 7
Changing the machine 8
Running, and stopping 9
Speed 10
RESET is not POWER 10
Every monitor command 12
Running the machine 12
Examining and changing memory 16
Debugging and tracing 22
Configuring the machine 26
Getting help and leaving 0

The monitor

This document is one of two short documents about how to use **`altairsim` itself**. `altairsim` is the program that you run, and the machines that it simulates are a different subject. This document is about **the monitor**. The monitor is the `altairsim>` prompt, where you start, stop, examine and change the machine. The other document, *The Debugger*, tells you what to do in the monitor when a guest does not work correctly.

Both documents are companions to the *User Manual*. The *User Manual* describes the hardware: the boards, the machines, the disks and the tapes. `altairsim` simulates the MITS Altair 8800 and its S-100 bus. If you do not know these, start with the first chapter of the *User Manual*.

The `altairsim>` prompt is **the monitor**. The monitor is the front panel of the machine, and it is also the debugger. In `altairsim`, the front panel and the debugger are one thing. The monitor can do all that the front panel of a real Altair could do, and much more. It sets breakpoints, single-steps and disassembles. It also shows you the bus: which board decodes each address, which board asserts each interrupt line, and where two boards decode the same address. *The Debugger* describes these debug tools. The debug tools are the main reason that `altairsim` exists.

The monitor operates the machine directly. It is not a *menu* that stands between you and the machine and lets you examine a fixed set of things. There is no debug mode to enter, and no process watches the machine from outside. A breakpoint is **the machine stopping**. It is not a script that sees afterward that the machine should have stopped.

`IN` and `OUT` do **real bus cycles**, with all the side effects of a real cycle. If you read a UART's data port in the monitor, you take the byte from the UART, as the guest would. The front panel and the debugger are one object because on an Altair they were one object. The operator set the switches and read the lamps.

The monitor works while the machine is stopped. Most of the tasks that follow work with the power on and the processor stopped. Examples are examining memory, doing a bus cycle and adding a board. The front panel was also made for a machine with the power on and the processor stopped.

Commands resolve by prefix

The monitor finds a command from the first letters that you type. It has **no aliases and no abbreviations to learn**. Type enough letters to make the command unambiguous. The first command that matches is the one that runs.

`HELP` prints the full list of commands, with the shortest form of each command outside the brackets:

```
BO[ARDS]          B[REAK]           COM[PARE]         C[ONFIG]
CONN[ECT]         CONS[OLE]         DE[POSIT]         DI[SASM]
DISC[ONNECT]      DO                D[UMP]            E[DIT]
EX[AMINE]         F[ILL]            HE[LP]            H[ISTORY]
I[N]              L[OAD]            MA[CHINE]         M[OUNT]
MOV[E]            N[EXT]            NO[BREAK]         O[UT]
P[OWER]           Q[UIT]            REGI[ON]          RE[GS]
RES[ET]           REST[ORE]         R[UN]             SA[VE]
SEA[RCH]          SE[T]             SH[OW]            SN[APSHOT]
STA[RTUP]         S[TEP]            SY[MBOLS]         T[RACE]
TY[PE]            U[NMOUNT]         W[HO]
```

Type the part before the bracket. `D` is `DUMP`, `DE` is `DEPOSIT` and `RES` is `RESET`. Upper case and lower case are the same, in a command and in the name of a board.

`HELP <command>` shows the usage and examples for one command. `?` is the same as `HELP`.

Some commands are shortcuts for a longer command. The help of a shortcut says which command it is short for. `CONSOLE` is a shortcut for `SET CONSOLE` and `SHOW CONSOLE`. Thus `HELP CONSOLE` prints the full help of `SET CONSOLE`, which is also what `HELP SET CONSOLE` prints.

> **`R` is `RUN`, not `RESET`. To reset, type `RES`.** You type `RUN` in every session, and an unwanted `RUN` does no damage. If `R` reset the machine, one wrong key could make you set up the machine again.

Editing the command line

The monitor has a full line editor. The key with the label Backspace erases the character to the left of the cursor. It does this for each byte that a terminal can send for that key. The arrow keys move the cursor in the line. The line editor also keeps a **command history**. The up arrow goes back through the lines that you typed, and the down arrow goes forward to the line that you were typing.

The history is **saved between sessions, for each directory**. When you quit, the monitor writes your last commands to a hidden file, `.altairsim_history`, in the directory where you started `altairsim`. The next time you start `altairsim` in *that* directory, the up arrow shows those commands. Each project directory keeps its own list. The monitor writes the history file only when you type at a real terminal. A script, a pipe or an automated run never makes a history file.

The `history` setting on the console sets how many lines the history file keeps. The default is 50. Type `SET CONSOLE history=200` to keep more lines. Type `SET CONSOLE history=0` to stop saving the history

file.

Completing with `Tab`

`Tab` completes the word that you are typing. It gets the possible words from the current machine. When you add a board, `Tab` can complete the board's id at once. There is no list to keep up to date. `Tab` completes these words:

- at the start of a line, a **command**: `SH⇥` → `SHOW`
- after `SET`, a **board id**, or `CONSOLE` or `DISPLAY`: `SET me⇥` → `SET mem0`
- after a board id, one of **the board's property names**, with the `=` added: `SET mem0 fi⇥` → `SET mem0 fill=`
- after the `=`, one of **the property's legal values**: for `SET mem0 fill=⇥` the choices are `zero` and `random`

When more than one word fits, `Tab` adds the letters that all of the words share, and then stops. Press `Tab` again to see the list. When no word fits, `Tab` does nothing.

Editing keys

Key	Does
`←` `→`	move one character
`Ctrl-A` / `Home`	move to the start of the line
`Ctrl-E` / `End`	move to the end of the line
`Alt-B` / `Ctrl-←`	move back one word
`Alt-F` / `Ctrl-→`	move forward one word
`Backspace`	erase the character before the cursor
`Delete`	erase the character under the cursor
`Ctrl-W`	erase the word before the cursor
`Ctrl-K`	erase from the cursor to the end of the line
`Ctrl-U`	erase the full line
`↑` `↓`	move back and forward through the command history
`Tab`	complete the word at the cursor
`Ctrl-D`	on an empty line, quit, the same as `QUIT`

> **In the monitor, `Ctrl-E` moves to the end of the line.** In the monitor you type to the *line editor*. When a **running** guest has the console, `Ctrl-E` is **STOP**. `Ctrl-E` stops the machine and starts the monitor, so you have the keyboard again (see below). The same key does two jobs in two places.

Repeating the last command: `.`

Type `.` alone on a line to run your **last command again**. The monitor does not echo the command, so you see only its output. The commands that you usually repeat continue from where they stopped. A bare `DISASM` disassembles the next screen, a bare `DUMP` shows the next page, and `STEP` steps again. To go forward through a routine, type `DI` one time and then `.` `.` `.`. To single-step, type `S` and then `.`.

Each `.` repeats the original command, not the previous `.`. Ten presses of `.` run the original command ten times. If you type `.` before any other command, the monitor tells you that there is nothing to repeat.

Reaching the host: `!`

A line that starts with `!` goes to **your host shell**. The monitor sends all the text after the `!` to the shell with no changes, spaces included. The monitor waits until the command ends, and then it shows the `altairsim>` prompt again:

```
!ls                list the directory you started from
!vi HELLO.PRN      open a file in your editor, then :q to return to the monitor
!cp game.dsk save.dsk   copy a disk and keep it mounted
```

`!` runs **your** shell, with your own permissions. The shell is not part of the machine, and the guest cannot see it or use it. An editor works because the monitor releases the keyboard before it starts the command. `vi` gets a normal terminal, and it gives the terminal back when it exits. The machine stays stopped while the command runs, as it always is while the monitor runs.

A bare `!` with no command shows how to use `!`.

Numbers: one fixed rule

> **On the wire → hex. Never on the wire → decimal.**

If the processor can see a number, the number is **hex**. Addresses, ports, data bytes and registers are hex. If the processor never sees a number, the number is **decimal**. Counts, widths, sizes and drive numbers are decimal.

Hex is only the *default* for the wire class. If you set the console to octal, the monitor reads and prints the wire class in octal. The MITS manuals and the front panel used octal. The rule stays the same. Octal replaces hex for the wire class, and counts stay decimal. *Reading and writing in octal*, below, tells you how to use octal.

```
DUMP 100            address  -> 0100 hex
STEP 10             a count  -> ten instructions
OUT FF 55           port and byte -> both hex
SET sio0:a baud=9600      a baud rate -> nine thousand six hundred
```

A prefix or a suffix sets the base of one number:

Write	Base
`0x` , `$` , or a trailing `h`	hex
`0o` , or a trailing `q`	octal
`%`	binary. Use it for the front panel's sense switches: one digit for each switch.
`0b`	binary, but only where the number is decimal or octal. Where the number is hex, `B` is a digit: `0B10` is the address 0B10.
a leading `#`	decimal
a `K` or `M` suffix	**always** decimal. `48K` is 49,152.

The monitor rejects a number with two bases, such as `0x10K` . It does not guess.

The same rule applies in the monitor, in a machine file and in the settings of every board. You learn only one rule.

The **classes** are fixed. Each kind of number is always in the same class. You can change the base in which the monitor *prints* the wire class, and the next section tells you how to change it.

Reading and writing in octal

The MITS manuals and the Altair front panel used **octal**, not hex. You can use octal too:

```
SET CONSOLE base=octal
```

After this command, the wire class is **octal**. The monitor reads and prints addresses, ports, data bytes and registers in **split octal**. Each byte is a group from `000` to `377`, and a 16-bit address is two groups. The front-panel address lamps are grouped the same way:

```
EXAMINE 100         -> 000 100  076   (the byte 0x3E at address 0x40)
DUMP 100-100        -> 000 100  076
DISASM 0            -> JMP 022 064     (a jump to 0x1234)
```

A bare number is octal now too, so `100` is address `0x40`. The decimal class, such as counts, widths and baud rates, does not change. The prefixes and suffixes still work in both modes, so you can always type a number in the base that you want. `0x1234` is hex in octal mode, and `0o377` is octal in hex mode. `base=hex`, the default, sets hex again. To start in octal every time, set `[console] base = octal` in a machine file.

Naming a board: `<id>[:<unit>]`

Every board in the machine has an **id**. The id is a name that you give the board when you add it, in a machine file or with `BOARDS ADD`. An id can be any name, such as `sio0` or `serial`. Upper case and lower case are the same.

The machines that come with `altairsim` use a short type name and a number, such as `cpu0`, `sio0` and `dsk0`. The number has no special meaning. It only tells two boards of the same kind apart, such as `sio0` and `sio1`.

Some boards also have **units**. A unit is a part of a board that has its own name. For example, a 2SIO serial board has two channels, `a` and `b`. A floppy controller has four drives, `drive0` to `drive3`. A memory board has a ROM socket, `rom0`. To name a unit, type the board id, a colon and the unit name:

```
SHOW sio0                  the board
SET  sio0:a baud=1200      channel a of the board
MOUNT dsk0:drive1 my.dsk   drive1 of the board
```

Shorter names

When you type in the monitor, you can shorten the id, and you can omit the unit.

You can omit the digits at the end of an id. `sio` finds `sio0`, and `dsk` finds `dsk0`. A shorter id works only when one board has that id with digits after it. If the machine has `sio0` and `sio1`, `sio` is ambiguous. The monitor then lists `sio0` and `sio1` and stops. You can omit only the digits at the end, so `si` does not find `sio0`.

You can omit the unit when the board has only one unit that the command can use. For `MOUNT`, the monitor counts only the units that take a medium, such as a drive or a tape. For `CONNECT`, it counts only the serial units. For example, `MOUNT acr tape.bin` puts a cassette in the one tape unit of `acr0`. A floppy controller has four drives, so `MOUNT dsk0 my.dsk` does not work. The monitor lists the four drives and stops.

To change a property of a unit, `SET` always needs the unit name. Without a colon, `SET` changes a property of the board.

Shorter names work only in monitor commands. A machine file, and an MCP tool that takes a board id, must use the full id and the full unit name.

Seeing the machine

```
altairsim> BOARDS
  ID    TYPE        I/O       UNITS                                                                  MEMORY
  ----  ----------  --------  ---------------------------------------------------------------------  ------------
--------------------
  fp0   fp          FF        -                                                                      -
  cpu0  8080        -         1 cpu: 8080                                                            -
  sio0  2sio        10,12     2 serial: a*, b                                                        -
  dsk0  dcdd        08,09,0A  4 disk: drive0(empty), drive1(empty), drive2(empty), drive3(empty)  -
  hb0   hostbridge  B0,B1     -                                                                      -
  mem0  memory      -         1 rom: rom0                                                            0000-DFFF
ram  56K
                                                                                                     FF00-FFFF
rom  dbl  phantom:all

  * holds the console
```

The `BOARDS` output is the backplane. It shows each board in the machine, the ports and memory that the board decodes, and what is in its units.

Command	Shows
`BOARDS`	the backplane
`SHOW <id>`	one board: each setting, its value, and the values that it accepts
`SHOW MACHINE`	the whole machine
`SHOW CONSOLE`	which unit has your keyboard, and which transforms apply to its bytes
`SHOW DISPLAY`	the windows altairsim opens: whether a window comes to the front when it opens, where its keys go, and whether the CRT look is on
`SHOW JOYSTICKS`	the host game controllers that a board can read, such as a D+7A (needs an SDL3 build)
`SHOW SERIAL`	the serial ports on your computer, and the unit that holds each port
`SHOW BUS`	the whole backplane at one time: `MEMORY`, `I/O` and a short `INTERRUPTS` summary
`SHOW BUS MAP`	which board decodes each address, and which addresses float
`SHOW BUS IO`	which board decodes each port
`SHOW BUS IRQ`	which board is strapped to each interrupt line, and which board asserts it
`SHOW BUS CONTENTION`	where two boards decode the same address or port

Look closely at `SHOW <id>`, because it is the **only** command that you need to configure a board. It lists every property, its value and its legal values. The property names **are** the keys that you write in a machine file. `altairsim` has no second schema. The board reference in the *User Manual* is printed from the same tables. (The reference at the end of this document is for the monitor's commands.)

Changing the machine

```
SET cpu0 clock_hz=2000000      run at the real 2 MHz clock
SET mem0 fill=zero             RAM starts as zeros, not random bytes
SET fp0  sense=80              set the SENSE switches
BOARDS ADD 2sio sio1 port=20   add a second serial board
BOARDS REMOVE sio1             remove it
CONFIG SAVE mine.toml          save the machine as it is now
```

`altairsim mine.toml` starts the same machine that `CONFIG SAVE` saved.

Running, and stopping

```
RUN FF00     set the PC and start: the same two steps as on the front panel's switches
RUN          continue from where the processor stopped
```

`RUN <addr>` is EXAMINE followed by RUN, as on the front panel. `altairsim` has no `BOOT` command, and this is deliberate. A machine starts only when a `RUN` tells it to start, as a real Altair started only when the operator pressed RUN.

If a board holds the console, **the guest gets the keyboard**. The guest gets every key, including `Ctrl-C`, because a CP/M program can read `Ctrl-C`.

`Ctrl-E` stops the machine and starts the monitor

`Ctrl-E` is the front-panel STOP switch. `altairsim` reads `Ctrl-E` *before the guest can see the byte*. The guest never gets `Ctrl-E`, and it cannot disable `Ctrl-E`. `Ctrl-E` always stops the machine.

If the guest needs `Ctrl-E`, move STOP to a different control key. For example, type `SET CONSOLE stop=1D` to use `Ctrl-]`. The value is the key's control code, from `01` (`Ctrl-A`) to `1F` (`Ctrl-_`). To use the new key every time, set `[console] stop = 0x1D` in a machine file.

`Ctrl-E` stops the machine and starts the monitor. The guest no longer has the keyboard, and you type to the monitor. No instruction executes while the monitor runs.

```
A>
STOP -- the machine is still at CA9C. RUN resumes.
altairsim>
```

STOP does not *change* the machine, because STOP is not RESET and not POWER. The registers, the memory and the disks stay as the guest left them. A bare `RUN`, with no address, continues at the instruction that the processor was about to execute. The message "*still at CA9C*" tells you this.

How a run ends

A `RUN` ends when it reaches a **breakpoint**, or a `HLT` that no interrupt can end. It always tells you which. If no board holds the console, the guest does not get the keyboard. To stop that run, press `Ctrl-C`.

After a `HLT`, the processor stays halted until `RESET`. On a halted processor, `RUN`, `EXAMINE` and `DEPOSIT` do nothing, as the front-panel switches do. The monitor tells you: "*processor halted at PC=0101 -- RESET ends the halt.*" *The Debugger* has the details.

`RUN` and `Ctrl-E` are the front panel's RUN and STOP switches. `RUN` starts the processor. STOP, a breakpoint or a `HLT` stops the processor and starts the monitor. **The monitor is available only while the machine is stopped.** While the machine runs, the guest has the keyboard and there is no `altairsim>` prompt. Every `SET`, `DEPOSIT` and `EXAMINE` acts on a *stopped* machine. A property never changes during an instruction. No property is locked while the machine runs, and no property can be set only while it runs.

Speed

By default, the machine runs as fast as the host can run it. `clock_hz` on the CPU board is `0`. For example, a cassette that took 110 seconds on a real Altair loads in about one second.

Type `SET cpu0 clock_hz=2000000` to run at the speed of a real 2 MHz Altair. The guest sees the same result at either speed, because the cassette uses the same number of T-states. The real clock speed changes how the machine *feels*, not how it *behaves*. `SHOW cpu0` shows `achieved_hz` next to `clock_hz`. `achieved_hz` is the clock speed that the run loop reached. You can read it, but you cannot set it.

A crystal is always kept: at the prompt, from a script, from a pipe and under `--mcp`. A script that must be fast leaves `clock_hz` at 0, or starts with `SET cpu0 clock_hz=0`. `SHOW CLOCK` tells you which: its `pacing` line reads `paced` or `free`.

The speed matters to a guest that measures time against the *outside* world. An XMODEM transfer needs the real clock speed, and a cassette does not. For more information, see `clock_hz` and `idle` in the Boards chapter of the *User Manual*, and see its Troubleshooting chapter.

RESET is not POWER

`RESET`	the bus's RESET* line. The processor restarts at `0000`. **Memory stays the same**, and the disks stay mounted.
`POWER`	a power cycle. **This is the only command that loses the contents of RAM.** It also reads the ROM images again.

`RESET` does not clear memory, because RESET on a real Altair did not clear memory. Much period software needs this behavior.

`RESET*` is a **line on the backplane**. It is not an instruction that `altairsim` does for you. Every board receives it, and each board does what its real hardware did. The result is different for each board. The memory board clears its bank latch, but it does not change RAM, because a RAM chip has no reset pin. The floppy controller flushes the sector that it was writing and deselects the drive.

The 2SIO does **nothing**, because the 6850 has no reset pin for `RESET*`. Its baud rate, word format and interrupt enables stay the same after a reset, as on real hardware. If you type `RESET` during a disk write, you get the result that the hardware gave. You get a half-written sector and a serial port that is still set as the failed guest left it. All of RAM stays the same.

`POWER` uses a different line. When the machine powers on, it drives `POC*` (Power-On Clear), which is a separate backplane line. A board can respond to `POC*` and `RESET*` in different ways, because the real cards did. The 88-VI/RTC is an example. `POC*` disables the board, and `RESET*` is not connected to it. If a crashed guest left the interrupt controller armed, the controller **stays armed after a `RESET`**. Only `POWER` clears it.

`POC*` is also the only time when RAM loses its contents. On `POWER`, the memory board fills RAM again, with **random bytes by default**, because static RAM does not start with zeros. It also reads every ROM image again.

	The processor	**The boards**	**RAM**
`RESET`	restarts at `0000`	`RESET*` on the bus. Each board does what its hardware did, and some do nothing.	**stays the same**
`POWER`	restarts at `0000`	`POC*` on the bus. Each board starts as it did at power-on.	**filled again**, ROMs read again

Every monitor command

Commands resolve by prefix, and the first match wins. There are no aliases and no fixed abbreviations: the shortest prefix that reaches a command is derived from the table's order, so it is shown here as `D[UMP]` — type the part before the bracket. (`?` is the one true alias, for `HELP`.)

`.` **repeats your last command.** It runs quietly, with no echo, so a single keystroke walks the continuing verbs forward: `DI` disassembles a screenful, then `.` `.` `.` keeps going; the same holds for `DUMP` and `STEP`.

Numbers: on the wire is **hex** (addresses, ports, bytes); never on the wire is **decimal** (counts, widths, sizes). `0x` / `$` / `h` force hex, `0o`/trailing-`q` force octal, `#` forces decimal, and a `K` / `M` suffix is always decimal. `SET CONSOLE base=octal` makes the wire class read and print in **split octal** (each byte its own `000`–`377` group, an address as two of them) — the MITS front-panel convention; `base=hex` is the default. Either way both spellings stay typeable.

The commands are **grouped by what they do**, and within a group they are in **alphabetical order**. The abbreviation beside each name is still derived from the master table's priority order, not from this listing, so it is what you type regardless of where the command sits here.

Running the machine

NEXT — `N[EXT]`

```
NEXT
```

STEP that does not descend. A CALL or RST runs to completion and stops at the return address instead of stepping into it; anything else is a plain single step. It is a temporary breakpoint at the return plus a RUN, so the callee is LIVE -- it can use the console, and ^E (STOP) or ^C stops it. On a HLT that nothing can end, NEXT does nothing. RESET ends the halt.

```
N  ; over the CALL/RST at PC (else single-step)
```

NOPASTE — `NOP[ASTE]`

```
NOPASTE
```

Stop a PASTE that is not finished. The part of the file that the guest has not read is dropped, with any TYPE that waited behind it, and the count is shown.

```
NOPASTE
```

PASTE — `PA[STE]`

```
PASTE <file>
```

Send a host file to the guest's keyboard, as if you pasted it. TYPE sends a short text; PASTE sends a file of ANY SIZE. The file is read as the guest takes the characters, so no character is lost and the file is never held whole. At the end of the file the keyboard is yours again.

The bytes go as they are in the file -- no line ending is changed. The guest reads them when it looks at the keyboard, so PASTE before the RUN that starts the guest is read at its first prompt. A TYPE after a PASTE waits for the file to end, so the two arrive in the order you gave them.

A relative is found beside the machine file, the same as LOAD.

This puts R.HEX on a CP/M disk. In CP/M, start PIP R.HEX=CON: and press ^E. Then:

```
PASTE "hostbridge/R.HEX"
RUN
```

When PIP has shown the last line, press Ctrl-Z to close the file.

The keyboard comes first: a key that you press during a paste goes to the guest before the remainder of the file. ^E stops the machine during a paste, and RUN continues it. NOPASTE drops it.

POWER — `P[OWER]`

```
POWER
```

Power cycle. THE ONLY THING THAT LOSES RAM -- a RESET does not, because on real hardware it does not.

RESET — `RES[ET]`

```
RESET [CPU]
```

A reset does NOT clear memory. Only removing power does that -- see POWER. A reset ends a HLT: the PC goes to 0000, and the HLTA lamp goes out. RESET CPU is a debugging convenience, NOT a real signal: no wire on the backplane resets the processor and nothing else.

RUN — `R[UN]`

```
RUN [addr]
```

Start the machine. `RUN <addr>` is EXAMINE + RUN -- it loads the PC first, exactly as you would on the panel.

If a unit holds the console, the GUEST GETS THE KEYBOARD -- every key, including ^C, which a CP/M program is entitled to read. The way back is STOP (^E), which the host takes before the guest is ever offered the byte, so the guest cannot disable it. STOP halts the machine -- nothing executes while this prompt is up -- but it does not DISTURB it: it is not RESET and not POWER, so every register, every byte and every disk survives, and a bare RUN resumes at the exact instruction. Stopped is not lost, and the debugger is at its most useful here: REGS, EXAMINE, DUMP, DISASM and STEP all work at this prompt.

IT RUNS FLAT OUT unless the CPU board has a crystal. `clock_hz` defaults to 0, so a cassette that took a real Altair 110 seconds comes off in about one. `SET cpu0 clock_hz=2000000` buys back the 2 MHz machine AND its 110 seconds. What the guest sees is identical either way -- the tape still costs the same T-states -- so the crystal buys period FEEL, not period behaviour.

A CRYSTAL IS ALWAYS KEPT: at this prompt, from a script, from a pipe and under --mcp. A script that must be fast leaves `clock_hz` at 0, or starts with `SET cpu0 clock_hz=0`. SHOW CLOCK says which: `paced` or `free`.

With no console connected there is no keyboard to hand over: it simply runs, ^C stops it. Either way it stops on a breakpoint or on a HLT nothing can wake, and it ALWAYS says which.

A HALTED PROCESSOR fetches nothing, so RUN does nothing until RESET. RUN refuses any halt, because it cannot move the PC. A bare RUN refuses a HLT that nothing can end. It continues a HLT that waits for an interrupt (EI, then HLT), and the interrupt ends the halt.

```
RUN F800  ; boot the monitor PROM
RUN       ; carry on from wherever the PC is
```

STEP — `S[TEP]`

```
STEP [n]
```

One instruction, with REAL bus cycles through the real decode. Prints one line per instruction; past 32 it runs quietly and reports. `n` is a count, so it is decimal. On a HLT that nothing can end, STEP does nothing. RESET ends the halt.

A LINE IS THE STATE AFTER THE INSTRUCTION RAN -- the registers as they now stand, and the instruction the PC has reached next. So `S 3` prints three lines, one per step, and the last line is where the monitor has left you: the next instruction, not yet run.

```
S     ; one instruction
S 10  ; ten of them
```

```
altairsim> DEPOSIT 0 3E 05 06 0A 80 76  ; MVI A,5 / MVI B,0A / ADD B / HLT
altairsim> EX 0
altairsim> S 3
C0Z0M0E0I0 A=05 BC=0000 DE=0000 HL=0000 SP=0000 IE=0 PC=0002  MVI B,0A
C0Z0M0E0I0 A=05 BC=0A00 DE=0000 HL=0000 SP=0000 IE=0 PC=0004  ADD B
C0Z0M0E1I0 A=0F BC=0A00 DE=0000 HL=0000 SP=0000 IE=0 PC=0005  HLT
```

Three instructions ran, three lines. Each shows the result: A=05 lands on the line for the MVI that loaded it, and the last line is the HLT waiting, not yet run. The flags are the 8080's own five, in the Altair's lettering -- Carry, Zero, Minus, Even parity, Interdigit carry -- and E goes to 1 on the ADD because 0F has an even number of bits set.

TYPE — `TY[PE]`

```
TYPE "text"
```

Put characters in the console's input buffer as if they were typed at the guest -- the same keystrokes the VDM window or a terminal would send. The guest reads them when it next looks at the keyboard, so they are TYPE-AHEAD: they wait in the buffer and are not forced on a program that is not reading.

Escapes: \r carriage return, \n line feed, \t tab, \ backslash, " quote. ENTER sends a carriage return, so a command line for the guest ends in \r.

Two more escapes send a key that you cannot put in the text. ^X is Ctrl-X: ^C is Ctrl-C, ^Z is Ctrl-Z, ^[is ESC and ^? is DEL. \xHH is the byte with the hex value HH, and it must have two digits.

```
TYPE "PRINT \"HI\"\r"   ; a line with quotes in it
TYPE "\^C"               ; Ctrl-C, to stop a BASIC program
```

This is how a machine file starts a program the monitor cannot reach. `startup` runs MONITOR commands; `XE TRK80` is input to SOLOS, a program INSIDE the machine. Put TYPE before the RUN that

starts the guest and the guest reads it at its first prompt -- in a TOML the backslash doubles (\r), because the config parser keeps one for the command:

```
startup = ["MOUNT sol0:tape1 \"TRK80.WAV\"", "TYPE \"XE TRK80\\r\"", "RUN C000"]
TYPE "XE TRK80\r"  ; type it now, at a running guest
```

A program that clears its keyboard as it starts drops keystrokes sent before it is ready; TYPE cannot help there, no more than a fast typist could.

TYPE is for a line or two. To send a whole file, use PASTE.

Examining and changing memory

COMPARE — `COM[PARE]`

```
COMPARE <range> <addr>
```

Byte for byte, a against the SAME LENGTH starting at -- memory to memory, both in the machine's address space. Every mismatch prints both addresses and their bytes; then a total. It changes nothing and runs no cycle.

```
COMPARE 0-FF 200        ; page 0 against page 2
COMPARE FF00-FFFF E000  ; the boot PROM against a copy up at E000
```

DEPOSIT — `DE[POSIT]`

```
DEPOSIT <byte> | DEPOSIT NEXT <bytes...> | DEPOSIT <addr> <bytes...>
```

The front-panel switches DEPOSIT and DEPOSIT NEXT. The PC is the address, as on the panel:

```
DEPOSIT <byte>             ; writes the byte at the PC. The PC does not move.
DEPOSIT NEXT <bytes...>    ; for each byte, adds 1 to the PC, then writes.
DEPOSIT <addr> <bytes...>  ; loads the PC with <addr>, as EXAMINE does,
                           ; writes the first byte, then does DEPOSIT NEXT
                           ; for each byte after it.
```

The PC stops on the last byte written. To run what you wrote, give its start: RUN , or EX and then RUN.

Each write is a REAL bus write. If no board decodes the address, the byte is gone, and DEPOSIT says so. A byte is 00-FF. A larger value is an error, and DEPOSIT writes nothing.

DEPOSIT needs a CPU, as the panel does. On a machine with no CPU, use EDIT, FILL or LOAD. While the processor is halted, DEPOSIT does nothing. RESET ends the halt.

```
DE 100 C3 00 F8  ; JMP F800 at 0100. The PC is now 0102
DE NEXT 00       ; 00 at 0103
DE 76            ; HLT at the PC (0103)
```

DUMP — `D[UMP]`

```
DUMP [<addr>|<range>] [WIDTH=16]
```

Hex and ASCII. A bare address runs to the END OF ITS PAGE, and a bare DUMP continues from there -- so the rows and the columns both stay page-aligned however you first landed. WIDTH is a count, so it is decimal.

```
D 100        ; 0100-01FF, a whole page
D 0001       ; 0001-00FF: stops on the boundary, last line full
D            ; the next page
D FF00-FF0F  ; an explicit range means exactly what it says
D 100/20     ; 0100-011F (LEN is part of the address expression: hex)
D 0 WIDTH=8  ; eight bytes per line
```

EDIT — `E[DIT]`

```
EDIT <addr> [ROM]
```

Interactive DEPOSIT. The prompt shows an address and the byte that is there; type a new value and Enter writes it and drops to the next byte, bare Enter leaves it and drops to the next, and '.' returns to the monitor. Runs REAL bus writes, so it says so if no board decodes the address; ROM burns instead, the way LOAD ... ROM does -- behind the bus, into the chip that answers there. If the machine has a CPU, type an INSTRUCTION where a byte would go and it is assembled in place -- the prompt then drops by the instruction's length, not one byte. Operands are numbers in the console base (an H or Q suffix overrides); a bare value is still a plain byte. The 8080 and 8085 assemble in full; the Z80 is a convenience -- its documented main, CB and ED forms, but not the IX/IY indexed forms (they carry a displacement) or the relative JR/DJNZ jumps (their byte is a signed offset from the next address), which report "not

implemented" -- deposit those bytes directly, or use JP. Look at four bytes, patch two instructions in, and read them back:

```
altairsim> EDIT 100
0100 C3 IN 10       ; assembles DB 10, on to 0102
0102 00 LXI H,FF13  ; assembles 21 13 FF, on to 0105
0105 76 .           ; '.' returns to the monitor
altairsim> DISASM 100 2
0100  DB 10     IN 10
0102  21 13 FF  LXI H,FF13
(needs an interactive or piped session -- with none, use DEPOSIT)
```

EXAMINE — `EX[AMINE]`

```
EXAMINE [<addr>]
```

One byte: hex, ASCII, and the bits as the panel's LEDs showed them. EXAMINE jams the address into the PROGRAM COUNTER, just like the panel switch, so `EX <addr>` also shows the register line and the disassembled instruction the PC now points at -- what the next STEP will execute. Bare EXAMINE is the panel's EXAMINE NEXT -- it steps one byte, quietly. Its cursor is its own; a DUMP does not move it.

```
altairsim> EX 100
0100  C3  .  11000011
A=00 ... PC=0100  JMP 0138
altairsim> EX  ; and the next byte, and the next
```

While the processor is halted, EXAMINE does nothing. The panel loads the PC by giving the processor a JMP, and a halted processor does not fetch it. RESET ends the halt. DUMP and DISASM still read memory while it is halted.

FILL — `F[ILL]`

```
FILL <range> <byte>
```

```
FILL 0-3FF 00
```

IN — `I[N]`

```
IN <port>
```

Runs a REAL IN cycle, with real side effects: an IN from a UART's data port consumes the byte and the guest never sees it. To look without touching, use WHO IO . Reports whether anybody actually answered.

```
I 10  ; port 10 -> FF   (nobody answered -- the bus floated it)
```

LOAD — `L[OAD]`

```
LOAD <file> [AT <addr>] [FORMAT=BIN|HEX|SREC] [ROM]
```

Put a file into memory. There are three kinds of file and they differ in ONE way -- whether the file knows where it goes.

```
HEX  Intel HEX. ASCII text, and it CARRIES ITS OWN ADDRESSES, so it needs
     no AT. Each line is one record: a ':', a length, the address, a type
     (00 data, 01 end-of-file), the bytes, and a checksum. Every checksum
     is verified and a bad one FAILS the load and names the record -- a
     half-loaded program is a miserable thing to debug.
       :10010000C300F8AF32004D3E0132014D76C9AA5509
       :00000001FF
        ^^^^^^^^^^                              ^^ checksum: the record
        |  |    |                                 sums to zero, byte-wise
        |  |    type 00 = data (01 = end of file)
        |  load address: these bytes go at 0100
        10 = sixteen data bytes follow
```

```
SREC Motorola S-records -- the Motorola-world counterpart to Intel HEX.
     ASCII text that CARRIES ITS OWN ADDRESSES too, so like HEX it needs no
     AT. Data records are S1/S2/S3 (2/3/4-byte addresses), the terminator
     S7/S8/S9. Every checksum is verified and a bad one FAILS the load.
       S1130100C300F8AF32004D3E0132014D76C9AA5505
       S9030000FC
```

```
BIN  A flat binary: bytes, and nothing else. It carries NO addresses, so
     it cannot say where it goes and AT is REQUIRED. Without one, LOAD
     refuses rather than guess.
```

WHICH ONE IS DECIDED BY THE FILE'S CONTENTS, not its name -- Intel HEX and S-records each announce themselves, and a .bin full of HEX text is still HEX. FORMAT= overrides that when the sniff is wrong; it always wins.

AT MEANS 'PUT IT HERE', for both kinds. On a HEX file it relocates: the file's FIRST data record lands at AT and everything else moves by the same amount, wrapping at FFFF (a file whose first record is F000, loaded AT 0, puts its F800 record at 0800). Without AT, a HEX file loads where it says.

ROM is the PROM burner. LOAD writes through the bus, so a ROM never takes it -- a bus write cannot program a PROM on real hardware either, and LOAD says how many bytes landed nowhere rather than half-loading in silence. ROM goes behind the bus, straight into whichever chip answers reads at that address: the operator pulling the chip and putting it in a programmer. It is why the operator can write a ROM and the guest cannot.

```
LOAD dbl.hex                  ; where the file says, through the bus
LOAD monitor.bin AT F000 ROM  ; a flat binary, burned into the ROM there
LOAD prog.hex AT 100          ; relocate: first record goes to 0100
LOAD odd.txt AT 0 FORMAT=HEX  ; it IS hex, whatever it is called
LOAD mon.txt FORMAT=SREC      ; read it as Motorola S-records
```

MOVE — `MOV[E]`

```
MOVE <range> <dest> [ROM]
```

Copy a range of memory to . It reads the WHOLE range before it writes, so source and dest may overlap either way without a block eating its own tail. The writes are real bus cycles; ROM burns instead, the way EDIT and DEPOSIT do.

```
MOVE 100-1FF 200  ; page 1 up to page 2
MOVE 0-FFF 1000   ; the first 4K, up by 4K
```

OUT — `O[UT]`

```
OUT <port> <byte>
```

Runs a REAL OUT cycle. Says so if no board decodes the port.

```
O 10 41
```

REGS — `RE[GS]`

```
REGS | SET REG <r>=<v>
```

The flags are registers too, so SET REG CY=1 works. A register value is on the wire, so it is HEX.

What it shows is the ACTIVE CPU's own set: an 8080 prints A, the pairs, SP, PC and its flags on one line; a Z80 needs two -- the first carries that same set, the second the alternate bank (A' BC' DE' HL'), its shadow flags, IX, IY, I and IM. The register halves, R and F are reachable by name though they are off the line. SET

REG takes any name REGS knows -- and only those, so SET REG IX=0 needs a Z80. BREAK ... IF reads the very same names.

```
REGS
SET REG A=3F
SET REG PC=FF00
SET REG IX=8000  ; Z80 only
```

SAVE — `SA[VE]`

```
SAVE <file> <range> [FORMAT=BIN|HEX|OCTAL|PRN]
```

Memory to a file, through the bus -- so what you get is what the CPU would read, ROM included. The range is what to save; a byte nobody drives reads FF.

THE NAME DECIDES THE FORMAT, and this is the other half of LOAD's rule rather than the same one: LOAD can open the file and see what it IS, and SAVE cannot -- the file does not exist yet. So a name ending .HEX writes Intel HEX, .OCT an octal listing, .PRN (or .LST) a disassembly listing, and anything else a flat binary. FORMAT= says it outright when the name would guess wrong, and wins.

OCTAL and PRN are LISTINGS, not load formats: OCTAL is split-octal addresses and octal bytes, the way the MITS manuals and front panel showed memory; PRN is the DISASM listing -- address, object bytes, mnemonic and any SYMBOLS labels -- written to a file for reading and marking up. Both follow the console base. LOAD does not read either back: BIN and HEX round-trip, OCTAL and PRN do not.

```
SAVE out.hex 0-FFF             ; Intel HEX, by its name
SAVE out.bin F800-FFFF         ; a flat binary, by its name
SAVE out.oct 100-1FF           ; an octal listing, by its name
SAVE out.prn 100-1FF           ; a disassembly listing, by its name
SAVE out.dat 0-FFF FORMAT=HEX  ; hex, though it is not called .hex
```

SEARCH — `SEA[RCH]`

```
SEARCH <range> <bytes...>|"str"
```

```
SEA 0-FFFF C3
SEA 0-FFFF "CP/M"
```

Debugging and tracing

BREAK — `B[REAK]`

```
BREAK [<addr> | MEM R|W <addr> | IO R|W <port> | TAPE STOP] [IF <expr> | LOADS <expr>] [TRACE ON|OFF]
```

Bare BREAK lists them. Only the first kind is about the CPU at all -- MEM and IO watch BUS CYCLES, so they catch a DMA transfer too and work unchanged on any processor; TAPE STOP watches a DEVICE, halting when a cassette deck reaches its auto-stop mark -- the way to stop right after a load lands without knowing where the loader ends.

```
BREAK FF13       ; stop when PC gets there
BREAK 2C00-2CFF  ; ...anywhere in a range
BREAK MEM W 100  ; stop when anything WRITES 0100
BREAK IO R 10    ; stop on an IN from port 10
BREAK TAPE STOP  ; stop when a cassette deck auto-stops after a load
```

A breakpoint may carry a CONDITION and stop only when it holds. IF tests the registers. A bare word that names a register IS that register, so a literal is written with a leading zero (0A is ten, A is the accumulator). == != < > <= >= compare; && || combine; & | mask.

IF works on EVERY kind. On a plain BREAK it is the PC-arrival state. On a MEM or IO breakpoint it is judged at the instruction BOUNDARY, against the state the instruction began with -- its inputs -- so a conditional cycle breakpoint stops just AFTER the access, where an unconditional one stops just before it.

BREAK IO R also takes LOADS : like IF, but judged AFTER the IN retires, so it sees the byte the port just delivered. IF gates on the inputs; LOADS on the value read. (The Z80 reads a port into any register, or memory; name the one the IN targets. On an 8080 it is always A.)

The names are the ACTIVE CPU's own -- exactly the set REGS shows, every register and flag in it. On an 8080 that is A, BC/DE/HL, SP, PC and CY/Z/S/P/AC; a Z80 adds IX, IY, the alternate bank and its own flags. A name the running CPU does not have is an error, so IF IX==0 waits for a Z80 to be the one in the socket.

```
BREAK 100 IF A==0
BREAK 100 IF HL==8000 && Z==1
BREAK 100 IF (A&0F)==0
BREAK IO R 10 IF B==5     ; stop on an IN from 10, but only while B==5
BREAK IO R 10 LOADS A>7F  ; ...only when port 10 hands back a byte over 7F
BREAK 100 IF IX==8000     ; Z80 -- IX is not an 8080 register
```

TRACE ON|OFF makes it a TRACEPOINT: instead of stopping, it turns TRACE on or off and the machine RUNS ON. Two of them trace a REGION and nothing else -- which is how you trace one subroutine out of a program that would otherwise bury you. Like IF, it works on the MEM and IO kinds too. TRACE ON at an address traces the instruction AT it; TRACE OFF does not -- the region is [on, off).

```
BREAK 2C00 TRACE ON             ; start tracing when PC gets to 2C00
BREAK 2C40 TRACE OFF            ; ...and stop again at 2C40
BREAK MEM W 2000 TRACE ON       ; start when anything writes 2000 (that write is
                                ; the first line -- a trace shows its own reason)
BREAK 200 IF HL==8000 TRACE ON  ; conditional, and still does not stop
```

Where the trace GOES is TRACE's business, not the tracepoint's: set it up with TRACE ON MASK=..., then TRACE OFF to arm it without emitting. An unconfigured tracepoint traces to the console.

DISASM — `DI[SASM]`

```
DISASM [<addr>|<range>] [n] [CPU=8080]
```

It needs an INSTRUCTION SET, not a CPU -- so it works on an empty backplane. You normally never type CPU=: the active core says what it speaks, and DISASM asks it. It PEEKS, so it cannot consume a byte from a UART in the range.

n is how many INSTRUCTIONS to decode -- a count, so it is decimal, and 16 when you leave it off. It only applies to a start address: give a RANGE and the range decides where to stop. CPU= is an instruction set, one of 8080 or z80, and is only for when the machine has no CPU to ask.

```
DI FF00          ; sixteen instructions of the boot PROM
DI FF00 40       ; forty of them instead
DI               ; carry on from there
DI 0-2F          ; exactly that range
DI FF00 CPU=z80  ; decode as Z80 when nothing in the machine can say
```

HISTORY — `H[ISTORY]`

```
HISTORY [BUS|CPU] [n]
```

The last n INSTRUCTIONS the machine ran, oldest first -- a flight recorder that is always running while the machine runs, so it already holds the run-up to a breakpoint or a crash when you ask. Each line is exactly what STEP prints: the registers and flags as the machine stood, and the decoded mnemonic it was about to run. n is a count, so it is decimal; bare HISTORY shows the last 16.

The mnemonic is decoded from the bytes that ACTUALLY ran at that address, not from what the address holds by the time you look -- so code that rewrote itself (a DDT breakpoint going back, an overlay) reads truthfully. The line reflects the CPU that is in the socket now: on the twin-core card, switching cores makes earlier lines read in the new core's terms.

HISTORY BUS is the other recorder -- the raw BUS CYCLES, no registers and no mnemonics: T-STATE, TYPE (MR/MW a memory read/write, IN/OUT a port, INTA an interrupt ack), ADDR (or the I/O port), DATA, and then the two that make it a BUS trace rather than a CPU one -- who DROVE the cycle and who ANSWERED it, so a DMA transfer's cycles are in there and say whose they were. HISTORY CPU names the default out loud.

Each recorder is a FIXED ring of its last 8192: it overwrites its own oldest and never grows, so it costs the same whether the machine ran for a second or a week. Ask for more than 8192 and you get the 8192 it holds.

```
HISTORY          ; the last 16 instructions
HISTORY 100      ; the last hundred instructions
HISTORY BUS      ; the last 16 bus cycles
HISTORY BUS 100  ; the last hundred cycles
```

NOBREAK — `NO[BREAK]`

```
NOBREAK [id]
```

Bare NOBREAK clears them all. An id is not on the wire, so it is decimal.

```
NOBREAK 2
NOBREAK
```

SYMBOLS — `SY[MBOLS]`

```
SYMBOLS LOAD <file> [REPLACE] | SYMBOLS CLEAR
```

Load an assembler's symbols so you can BREAK, DUMP and EXAMINE by NAME instead of by hex, and SHOW SYMBOLS to read the table. Two file kinds, and one absolute rule:

```
.PRN / .LST   an assembler LISTING -- CP/M ASM, Microsoft M80, DR MAC. It marks
              an EQU, so a constant is told apart from a program label.
.SYM          the CP/M symbol file DR MAC/RMAC write and SID reads. A flat list
              of name=value, no label/constant distinction. (L80 writes no .SYM.)
```

ADDRESSES MUST BE ABSOLUTE. A relocatable M80 listing is refused, by the line -- link it and load the .SYM, or assemble to an absolute origin.

LOAD merges (the newest of a clashing name wins, and it says how many); REPLACE clears first; CLEAR empties the table. A file named in a machine's startup is reloaded on CONFIG LOAD and round-trips through CONFIG SAVE.

```
SYMBOLS LOAD prog.SYM
SYMBOLS LOAD roms/ALTMON/ALTMON.PRN
SYMBOLS CLEAR
```

TRACE — `T[RACE]`

```
TRACE ON|OFF [file] [MASK=IN,OUT,IRQ,DMA,CONTENTION]
```

Log every BUS CYCLE while the machine runs -- to the console, or to a file. A cycle, not an instruction: MR/MW are memory, IN/OUT are I/O, INTA is an interrupt acknowledge, and a granted DMA master's cycles are tagged [DMA]. This watches the same stream every board sees, so it is not a CPU feature and works unchanged on any processor.

MASK keeps only the cycles you name (no MASK keeps all): IN, OUT, IRQ, DMA, CONTENTION. A cycle is kept if it is any of them -- MASK=DMA is every cycle a master drove, whatever its type.

```
TRACE ON              ; every cycle, to the console
TRACE ON run.log      ; ...to a file
TRACE ON MASK=IN,OUT  ; just the port traffic
TRACE OFF
```

TRACE OFF stops the tracing but REMEMBERS where it was going -- a later TRACE ON, or a tracepoint, resumes to the same file and mask. That is what lets you aim a tracepoint at a file: TRACE ON run.log MASK=DMA, then TRACE OFF to arm it without emitting, then BREAK TRACE ON. See BREAK.

WHO — `W[HO]`

```
WHO <addr> | WHO IO <port>
```

Who WOULD answer -- it looks without running a cycle, so nothing is consumed and no board is poked. Reports contention, and reports PHANTOM*.

```
WHO FF00
WHO IO 10
```

Configuring the machine

BOARDS — `BO[ARDS]`

```
BOARDS [LIST]|ADD <type> <id> [k=v...]|REMOVE <id>
```

The backplane: what is in it, what each board answers to, and what is in its sockets. A bare BOARDS lists them. RAM and ROM are named separately, and a ROM range says which image is in it -- an empty socket decodes nothing, so it is not in the memory column at all; it is in UNITS, marked (empty).

```
BOARDS                  ; the backplane
BOARD                   ; the same thing: a prefix of BOARDS
BOARDS ADD memory mem0  ; fit one -- SHOW BOARDS lists the types
BOARDS REMOVE mem0      ; pull one out
```

CONFIG — `C[ONFIG]`

```
CONFIG LOAD <f.toml> | CONFIG SAVE <f.toml>
```

THE MACHINE, NOT WHAT IT IS DOING. SAVE saves the hardware you are actually running -- which boards, in what order, every property SET can write, what each unit is CONNECTed to, what is MOUNTed in each socket, and the startup list. It is the same format you would write by hand, and the same one a built-in is written in, so a saved machine is a first-class machine: LOAD it back, or name it on the command line, and you get exactly what you saved.

IT DOES NOT SAVE STATE, and that is not a gap to be filled: a machine file describes hardware, and none of this is hardware. NOT saved --

```
RAM             what you DEPOSITed is gone. LOAD/SAVE <file> <range> is for
                memory, and it is a separate file for a reason.
the registers   PC included, so a LOADed machine has not started.
breakpoints     nor tracepoints, nor where TRACE was pointed.
CONSOLE         stop and the transforms are the console's, not a board's in
                the backplane. They survive CONFIG LOAD untouched.
```

A SAVE IS A READ: it asks every property for its value and writes to nothing.

LOAD IS THE WHOLE MACHINE, so it REPLACES the one you have: the boards you had are out of the backplane, the new ones are in, and it is powered up and running its startup list -- a file whose startup

says RUN comes up running. Naming that same file on the command line does the identical thing; there is one road.

AND IT IS ALL OR NOTHING. The machine is built off to one side first, so a file that will not load -- a key that does not parse, a disk image that is not there -- leaves you exactly where you were. What you do not get back is the machine you REPLACED: there is no undo but the file you saved it to.

```
CONFIG SAVE machines/mine.toml
CONFIG LOAD machines/mine.toml  ; ...and this is how you get it back
```

CONNECT — `CONN[ECT]`

```
CONNECT <id>:<u> <endpoint>
```

PLUG IN THE OTHER END OF THE CABLE. A unit is a socket on the back of a board -- one of the 2SIO's two ports, say; an ENDPOINT is the thing at the far end of the cable, on the HOST side of the machine. It is not a board, it has no address, and the guest cannot see it: the 6850 clocks bytes the same way whether the wire ends at the console, a telnet session, a real RS-232 port, or nothing at all. No board in the machine knows what any of these words mean.

Endpoints: console | null | loopback | scripted | socket:PORT[?banner] | socket:HOST:PORT | telnet:PORT[?banner=off] | telnet:HOST:PORT | serial:DEVICE | pty[:LINK] | in:PATH | out:PATH |

terminal[?emulation=vt100&size=80x24] | printer:QUEUE | |FILE | |socket:PORT | |pty[:LINK] |

Endpoints: console | null | loopback | scripted | socket:PORT[?banner] | socket:HOST:PORT | telnet:PORT[?banner=off] | telnet:HOST:PORT | serial:DEVICE | pty[:LINK] | in:PATH | out:PATH |