altairsim — User Manual

2026-10-24 · 6f09484

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

What altairsim is 1
What is in the package 3
Running it 8
Quick start: CP/M in one command 10
Quick reference 14
Machines 27
Configuring a machine 39
Boards 55
Disks 106
Tapes 115
Serial ports, sockets and telnet 124
Moving files in and out 142
Worked examples 152
The MCP server 157
Troubleshooting 165
Glossary 172
Boards and their parameters 176
The built-in machines 234

What altairsim is

`altairsim` simulates the **MITS Altair 8800** and the **S-100 bus** that the Altair uses. A machine is a set of boards that plug into the bus. The program has built-in machines, and you can describe your own machine in a machine file.

The machine runs the software of the period as it was sold. Examples are Altair BASIC from a cassette, CP/M 2.2 from an 8″ floppy disk, and MITS Programming System II from paper tape. The software is not patched, and it cannot tell that the machine is simulated.

What you can do with it

- **Run period software.** Boot CP/M, load BASIC from a tape, and run the programs of the 1970s.
- **Write software for the machine, and find its bugs.** You can stop the machine at an instruction or a bus cycle that you choose. You can see which board answered and which board did not. When you give the machine the same input again, you get the same result.
- **Test a driver before you have the hardware.** Write a driver for a board that the program has, and run it against that board.
- **Connect the machine to your computer.** A serial board can use your terminal, a TCP socket or a real serial port. A line-printer board can print on the printer on your desk.
- **Let an AI assistant use the machine.** An assistant can boot the machine, type at it and debug a program through the MCP server.

What is in the program

- **Three processors:** the 8080, the 8085 and the Z80. The 8085 runs all 8080 code. Each processor gives the correct flags, including its undocumented behavior. Each one is tested with the standard processor exerciser programs.
- **Period boards:** processor boards, memory boards, serial boards, cassette interfaces, disk controllers, a line-printer controller and the front panel. The boards chapter describes each one.
- **Video boards.** The Altair had no video, but video boards were made for the S-100 bus. The VDM-1, the Dazzler, the SD Systems VDB-8024 and the Sol-PC are here. Each video board opens a window on your screen.
- **The monitor.** The `altairsim>` prompt controls the machine while it is stopped. From the monitor, you can set breakpoints, step one instruction at a time, disassemble, change memory, and trace the bus. You can also see which board decodes each address, and which boards answer the same address. *The Monitor* and *The Debugger* are two documents that ship beside this manual, and they describe these commands.

- **File transfer** between your computer and CP/M. A board that the period did not have moves the files, and the guest can use only the one folder that you choose. At the `A>` prompt, you type `HDIR`, `R` and `W`. These are ordinary CP/M programs, like `PIP` or `STAT`.

What it does not do

- **You can save the state of the machine, but you cannot replay a session.** `SNAPSHOT` writes the whole state of the machine to a file, and `RESTORE` reads it back. You cannot record a session and step backward through it.
- **There is no audio output.** The program reads and writes cassette `.WAV` files as files. The tapes chapter tells you how.
- **Not every S-100 board is here.** The boards chapter lists the boards that are.
- **The timing is correct for the software, but it is not a circuit simulation.** Each instruction takes the correct number of T-states. The program does not model propagation delay or analog behavior. No software from the period can see the difference.

Where to start

There is no installer, and there is nothing to set up. Unzip the package and run the program.

- To boot CP/M now, go to the *Quick start*. It takes one command.
- To learn what came in the package, read the next chapter.
- If the program does not start, read *Running it*.

What is in the package

```
altairsim                the program. One file, nothing to install.
QUICK-START.pdf          boot CP/M with one command. Start here.
altairsim-manual.pdf     this manual.
altairsim-changelog.pdf  what changed in this release, and in the releases before it.
altairsim-cheatsheet.pdf every command and option, formatted for reading.
altairsim-monitor.pdf    the altairsim> prompt: how to control the machine from the console.
altairsim-debugger.pdf   breakpoints, stepping, and how to look at the bus.
migrating.pdf            for users of AltairZ80 (SIMH) or z80pack.
DRIVING-WITH-AI.md       for an AI assistant that controls the machine. See below.
cheatsheet.md            the same reference as plain text, for an AI assistant to read.
LICENSE                  the MIT license of altairsim.
LICENSE-SDL3             the license of SDL3, which is built into the program.
LICENSE-MAME-HD63484     the notice for the source of the HD63484 model's structure (MAME).
examples/                machines that boot, with their media.
hostbridge/              the file-transfer utilities: source, HEX and COM.
tools/                   scripts that show a mirror, and scripts that open LBR files.
skills/                  the AI briefing again, and the rules for guest programs, as Agent Skills.
```

That is the whole package. You do not install a library or a runtime. You do not have to write a machine file before the program starts.

Start with `QUICK-START.pdf`. It boots CP/M with one command. It shows you the `Ctrl-E` key and the `RUN` and `QUIT` commands, which move you between the running machine and the monitor. That one page is enough to get a machine running. This manual gives the full detail.

If you used another Altair simulator before, read `migrating.pdf`. It tells you what carries over from **AltairZ80 (SIMH)** or **z80pack**, what has a different name here, and what you cannot do here.

`altairsim-changelog.pdf` is the release history. It tells you what each version added.

`altairsim-monitor.pdf` and `altairsim-debugger.pdf` are two more documents that ship beside this manual. This manual describes the simulated hardware. *The Monitor* describes the `altairsim>` prompt, where you start and stop the machine. *The Debugger* describes how to find a fault from that prompt. Each document tells you at the top where to start.

`altairsim` is one program that needs no other files. It uses one outside library, **SDL3**, to open the window for the video boards. SDL3 is built into the program, so its license is in the package as `LICENSE-SDL3`. `LICENSE-MAME-HD63484` is in the package for a similar reason. The model of the HD63484 video controller uses the structure of the MAME model, and the MAME license tells you to keep its notice with that code.

The **Developer Guide** is not in the package. It is with the source, at the address in What is not in the package: the source. You need it only if you want to build a board of your own.

`DRIVING-WITH-AI.md`

This document is for an **AI assistant**, not for you. It tells the assistant how to control the machine through the MCP interface of the program. To use it:

1. Make a working folder. If your assistant reads Agent Skills, install the skills as this section shows below. If it does not, put `DRIVING-WITH-AI.md` and the `skills` folder in the working folder. The document names files in `skills/`, which have the rules for programs such as `ASM` and MBASIC.
2. Start an assistant in that folder.
3. Ask for what you want, for example: "*Using altairsim, boot CP/M and show me what is on the disk.*"

If you do not use an AI assistant, you can ignore this document. Nothing else needs it.

`skills/` contains **Agent Skills**. Each folder in it is one skill. `skills/altairsim/` is the same briefing as `DRIVING-WITH-AI.md`. Each other folder, such as `skills/altairsim-mbasic/`, has the rules for one program that runs in the machine. Agent Skills come from Anthropic, for its Claude assistants. Claude finds a skill from its description, so when you ask for a machine, Claude reads the briefing itself. You do not have to tell it which file to open. To install the skills for Claude Code, copy every folder in `skills/` to one of these places:

- `.claude/skills/` in your project folder, for that project only
- `~/.claude/skills/` in your home folder, for every project

The folders need no other files, so they work in either place. Some other assistants can also read skills. Their documentation tells you where to put the folder. If your assistant does not read skills, give it `DRIVING-WITH-AI.md` and the `skills` folder as the steps above show.

The quick reference lists every option, every monitor command, every board and every machine. The program generated it, so it matches the program that you have. It ships in two forms with the same content. `altairsim-cheatsheet.pdf` is for you. `cheatsheet.md` is plain text, for an AI assistant to read.

The machines are in the program

You do not need any files to run a machine. The machine descriptions are built into the program, and when you give a name, that machine boots:

```
$ altairsim --list               what the built-in names are
$ altairsim altmon               a monitor in ROM, on a terminal
$ altairsim sol20                a Processor Technology Sol-20, running SOLOS
```

A built-in machine is an ordinary machine file that is stored inside the program. It uses the same TOML format that you would write yourself. `CONFIG SAVE mine.toml` saves any running machine to a file that

you can edit.

Some built-in machines have their software in ROM and need nothing else. Examples are `altmon`, `amon`, `sol20`, `vdm1`, `rombasic`, and the SD Systems `sbc200` and `sbc200v`. The other machines have at most a boot PROM, and they start with empty drives. They need media, which the next section is about. The machines chapter gives the full detail.

The examples, media included

`examples/` contains complete machines. **Each example is a folder with its media in it.** Every example boots as soon as you unzip the package. You do not have to get or mount anything.

Every folder has its own README, in Markdown and as a PDF. The README describes the example: what the machine is, what is in the drive or the tape reader, what to type, and what the machine should print. This manual does not list the examples. Look in `examples/`, and read the README of the example that you want.

```
$ ls examples/
$ altairsim examples/cpm/cpm22-buffered.toml
```

You can move an example folder anywhere. It still boots after you copy it, rename it or send it to another person. The machines chapter tells you how the paths in a machine file work.

The examples chapter describes some of the examples in detail. More examples, with their documentation, are at https://altairsim.com.

The file-transfer utilities

`hostbridge/` contains `R`, `W` and `HDIR`. The file-transfer chapter uses these programs to move files between CP/M and your computer. The folder has the 8080 **source**, and the assembled `.HEX` and `.COM` files.

You do not need this folder to use the utilities on the CP/M disk in the package. That disk already has the `.COM` files. The folder is for a disk that does not have the utilities. The file-transfer chapter tells you how to put them on that disk, by pasting `R.HEX` through the console.

The watch scripts

`tools/` contains two scripts that show the console of the machine in a second window, while an AI assistant controls the machine: `tools/mirror-watch.sh` for macOS and Linux, and `tools/mirror-`

`watch.ps1` for Windows. Each script waits for the mirror, shows its output, and connects again after the simulator starts again. `tools/README.pdf` gives the commands. The chapter *The MCP server* tells you how to start a mirror.

The LBR tools

Much CP/M software is kept in library files (`.LBR`). The files in a library are often crunched (a `Z` in the extension, such as `.AZM`) or squeezed (a `Q`, such as `.AQM`). `tools/` has three Python 3 scripts for these files. `tools/lbr-extract.py` copies the files out of a library. `tools/uncrunch.py` decodes a crunched file, and `tools/unsqueeze.py` decodes a squeezed file. Each decoder gives the file its original name. You must install Python 3 to use these three scripts. `tools/README.pdf` gives the commands.

What is *not* in the package: everything else to run

The media in `examples/` is all the media in the package. The other built-in machines that need a disk or a tape, such as `basic8k`, `ps2` and `minidisk`, start with an empty drive:

```
$ altairsim basic4k -x "SHOW MOUNTS"
altairsim> SHOW MOUNTS
  UNIT       KIND  HOLDS
  acr0:tape  tape  (empty)

  Paths are AS WRITTEN.  SHOW PATHS says what they are relative to.
```

You supply the media, and you `MOUNT` it. The disks chapter and the tapes chapter tell you how. When those chapters name an image that is not in `examples/`, the name shows the form of the command. It is not a file that you have.

Most disk and tape images are not ours to give away, so they are not in the package. More examples, with their media and their documentation, are at https://altairsim.com.

What is *not* in the package: the source

The **source code** of the simulator is not in the package. `altairsim` is an open project under the MIT license. You can get the source here:

https://github.com/deltecent/altairsim

The only source in the package is in `hostbridge/`. That is the 8080 source of the file-transfer utilities, not the source of the simulator.

You do not need the source to use anything in this manual. If you want to **build a board of your own**, you need the source and the *Developer Guide*. This manual tells you how to use the machine. It does not tell you how to add to the program.

Reporting a bug, or asking for something

Report a bug or ask for a feature in the **Issues** tab of the repository:

https://github.com/deltecent/altairsim/issues

Search the issues first. If nobody has reported your problem, open a new issue. You need a GitHub account.

Ask a question or talk about `altairsim` in the **Discussions** tab of the repository:

https://github.com/deltecent/altairsim/discussions

Use **Q&A** when you need help to do something, for example to boot a disk or set up a board. Use **Show and tell** to show a machine or a program that you made. An issue is for a bug or a feature request. If you open an issue that is only a question, we move it to Discussions.

A useful bug report lets another person see what you saw. Include:

- The **version**. This is the line that `altairsim` prints when it starts, or that `altairsim --version` prints. Paste the whole line. Between releases, the part after the version number tells which source built your copy. Also give your operating system. `SHOW VERSION` prints the same information from the monitor. It also has a `video` row that tells whether your copy can open a window. Include it in a report about a video board. The `tree` row tells whether the source had changes that were not committed when your copy was built.
- The **machine**. Give the name of the built-in machine, or paste the machine file. A machine file is a small text file.
- **What you typed and what happened.** Paste the terminal output, with the prompts. The monitor shows every command, so the pasted output is a full record.
- What you expected, when that is not clear.

If the guest software failed and the simulator did not, tell us which software it is and where you got it. Some period programs failed on real hardware too. With the image, we can find out which kind of failure it is.

A feature request is also an issue. Tell us what you are trying to do, not only the setting that you want. The machine often has a way to do it already. If it does not, your problem tells us what the answer should be. If you ask for an S-100 board that is not here, name the manual that documents the board. We build each period board from its manual.

Running it

`altairsim` is one program file. Unzip the package, and run the program from a terminal.

```
$ ./altairsim
AltairSim X.Y.Z -- 8080, full speed.
machine: default.  HELP for commands.
altairsim>
```

On **Windows**, the program is `altairsim.exe`:

```
> altairsim.exe
```

You can run the program from any folder. Put its folder on your `PATH`, and type `altairsim`.

The `altairsim>` prompt is **the monitor**. The machine has power, but it is stopped. No program runs until you start one.

macOS: the first run

The macOS program is signed with a Developer ID, and Apple has checked it (notarized it). macOS starts it with no message, also when you got the package with a browser.

The first time that you start a new copy, macOS asks Apple about the program. This check needs the internet. If the Mac is offline, macOS can refuse to start the program. The chapter *Troubleshooting* tells you what to do.

Getting help, and getting out

Type	To
`HELP`	list every command
`HELP DUMP`	show the usage and examples for one command
`QUIT`	leave the program

There is no `EXIT` command. Type `QUIT`, or only `Q`.

You can type a short form of a command. `HELP` shows each command with brackets, for example `D[UMP]`, `DE[POSIT]` and `RES[ET]`. The part before the bracket is the short form. You can type the short form, the full name, or any length between them.

Commands are not case-sensitive. Board names, board types, settings and their values are not case-sensitive either. The same is true for endpoints such as `socket:2323` and for command-line options such as `--mcp`. A file name, a host name and a serial device keep their case. This manual writes commands in capitals because they are easier to read.

Which machine you get

When you run `altairsim` with no arguments, you get the machine called `default`. It is a 56K Altair with a console, a floppy disk controller and a boot PROM. If the folder that you are in has a file called `altairsim.toml`, you get that machine instead.

To get a different machine, give its name or its file:

```
$ altairsim examples/cpm/cpm22-buffered.toml     a machine file: this one boots CP/M
$ altairsim basic4k                      a BUILT-IN machine, by name
$ altairsim --list                       what the built-in names are
```

A **built-in** machine is a machine file that is stored in the program. It uses the same format as the machine files in `examples/`. To see what is in a built-in machine, load it and look:

```
$ altairsim basic4k -x BOARDS
```

To get a built-in machine as a file that you can edit, type `CONFIG SAVE mine.toml`. This command writes the machine that you are running to a file, and that file boots.

The machines chapter describes every command-line option. It also tells you how `altairsim` decides if a word is the name of a built-in machine or the name of a file.

Quick start: CP/M in one command

This chapter boots CP/M 2.2 on a 56K Altair with an 8″ floppy disk. It also shows you the three commands that you use every time: `Ctrl-E`, `RUN` and `QUIT`.

Type the commands in the folder where you unzipped the package.

Boot CP/M

1. Start the machine:

   ```
   $ ./altairsim examples/cpm/cpm22-buffered.toml
   ```

   On **Windows**, the program is `altairsim.exe`, and the path uses backslashes:

   ```
   > altairsim.exe examples\cpm\cpm22-buffered.toml
   ```

   The machine boots CP/M:

   ```
   56K track-buffered CP/M 2.2b, booting off the 8" floppy in drive A.
   You land at the A> prompt -- type DIR to see what is on the disk.

   Drives B: through D: are empty. FORMAT one from CP/M, or MOUNT an image onto dsk0.
   AltairSim X.Y.Z -- 8080, full speed.
   machine: cpm22-buffered.  HELP for commands.
   startup> RUN FF00
   [console -- ^E returns to the monitor]

   56K CP/M 2.2b v2.3
   For Altair 8" Floppy

   A>
   ```

2. Type `DIR` to see what is on the disk:

```
A>DIR
A: L80      COM : LADDER   COM : ED       COM : ASM      COM
A: DUMP     COM : XSUB     COM : PCGET    COM : LS       COM
A: SUBMIT   COM : LOAD     COM : SURVEY   COM : VIEW     COM
A: LADDER   DAT : LUNAR    BAS : M80      COM : MAC      COM
A: MBASIC   COM : PIP      COM : STAT     COM : DDT      COM
A: MOVCPM8  COM : NSWP     COM : SYSGEN   COM : W        COM
A: ACOPY    COM : VI       DOC : STARTRK  BAS : TICTAK   BAS
A: VI       COM : CRC      COM : PCPUT    COM : AFORMAT  COM
A: STARINS  BAS : IOBYTE   TXT : R        COM : HDIR     COM
A>
```

   The disk has an assembler, a debugger, Microsoft BASIC, a text editor and Star Trek.

3. Type `MBASIC` to start Microsoft BASIC, and give it a line to run:

```
A>MBASIC
BASIC-80 Rev. 5.21
[CP/M Version]
Copyright 1977-1981 (C) by Microsoft
Created: 28-Jul-81
21816 Bytes free
Ok
PRINT 2+2
 4
Ok
```

4. Type `SYSTEM` to go back to the `A>` prompt.

Stop, continue and leave

1. Press **`Ctrl-E`**. The machine stops, and you get the monitor:

```
A>
STOP -- the machine is still at CA9C. RUN resumes.
C0Z1M0E1I0 A=00 BC=007F DE=CA01 HL=BC0E SP=BC37 IE=1 PC=CA9C  CALL CA78
altairsim>
```

2. Type `RUN`. The machine continues from the instruction where it stopped. Your `A>` prompt is where you left it:

```
altairsim> RUN
```

3. Press `Ctrl-E` again, and type `QUIT` to leave the program:

```
altairsim> QUIT
```

These are the three commands to remember:

`Ctrl-E`	Stop the machine, and go to the monitor. The machine loses nothing.
`RUN`	Start the machine again, from where it stopped.
`QUIT`	Leave the program. `Q` is enough. There is no `EXIT` command.

What `Ctrl-E` does

`Ctrl-E` does what the STOP switch on the front panel does. **The machine stops at the instruction where it was, and it loses nothing.** STOP is not RESET, and it is not POWER. The registers, the memory, the disk in the drive and the program counter do not change. While the machine is stopped, you can read memory, step one instruction at a time and set breakpoints.

`Ctrl-C` is not the stop key, because **`Ctrl-C` belongs to the guest**. CP/M does a warm boot when you press `Ctrl-C`, and BASIC stops the program. The program reads `Ctrl-E` before the guest can get it, and every other key goes to the guest. If a guest needs `Ctrl-E`, type `SET CONSOLE stop=1D` at the monitor. The stop key is then `Ctrl-]`.

Before you change the disk

Copy the folder before you write to the disk. The machine mounts the disk image read/write. Everything that you do in CP/M changes the file on your computer, and the program keeps no copy. The folder has all that the machine needs, and the copy boots from any place:

```
$ cp -R examples/cpm my-cpm
$ ./altairsim my-cpm/cpm22-buffered.toml
```

Go back to the `A>` prompt before you quit or copy the image. This BIOS keeps a full track in memory. It writes the track to the disk the next time it reads the console. If you quit before that, the last write is lost.

To look at a disk that you do not want to change, write-protect it at the monitor:

```
altairsim> MOUNT dsk0:drive0 cpm22b23-56k.dsk WP
```

The controller then refuses every write. CP/M does not know that the disk is write-protected, and a program that tries to write can fail. Do not use `WP` for a disk that you save work on. The disks chapter tells you more.

What happened when it booted

The machine file describes a **56K Altair with an 8″ floppy disk controller and a boot PROM at `FF00`**. It puts the disk image in drive 0, and it types `RUN FF00` for you. The `startup>` line shows this command.

There is no `BOOT` command. To boot a disk on a real Altair, you set the address switches to `FF00`, pressed EXAMINE, and then pressed RUN. `RUN FF00` does the same thing. A machine file can do only what you can type at the monitor.

The rest of the boot is real software. The PROM loads a boot loader from the disk. The boot loader loads CP/M into memory, and the BIOS of CP/M prints its banner.

The machine file names its disk `cpm22b23-56k.dsk`, with no folder. The program looks for the disk in the folder of the machine file. For this reason, the copy in `my-cpm` finds its own disk.

Where to go next

- **Other machines.** Type `./altairsim --list` to see the built-in machines. The machines chapter describes them.
- **A machine with no disk.** The tapes chapter loads Altair 4K BASIC from a cassette. You enter the bootstrap by hand, from the listing that MITS printed in its manual.
- **Complete sessions.** The worked examples chapter has more, from start to end.

Quick reference

Contents

- Getting out, and back in
- Editing the command line
- Command line
- Monitor commands
- Boards
- Machines
- A machine file, in one look
- Endpoints — `CONNECT <id>:<unit> <endpoint>`

Getting out, and back in

Key	Does
`^E`	**STOP** — stop the machine and take the keyboard back. Nothing is lost.
`RUN`	Resume, at the exact instruction it stopped on.
`QUIT`	Leave. (There is no `EXIT`.)

Editing the command line

`Tab` completes what you are typing — a command, then a board id, then its property names, then a property's values (`SET mem0 fill=` then `Tab`); a second `Tab` lists the choices. `Up` / `Down` walk the command history, saved per directory in `.altairsim_history`.

Key	Does
`Ctrl-A` / `Ctrl-E` (or `Home` / `End`)	start / end of line
`Alt-B` / `Alt-F` (or `Ctrl-Left` / `Ctrl-Right`)	back / forward one word
`Ctrl-W` / `Ctrl-K` / `Ctrl-U`	erase word behind / to end of line / whole line
`Backspace` / `Delete`	erase before / under the cursor

Command line

```
altairsim [machine] [options]

  machine            a built-in name, or a config file (has a '/' or ends .toml).
                     Omitted: ./altairsim.toml if there is one, else `default`.
  -m, --machine <n>  ALWAYS a built-in name -- never a file.
  -f, --file <path>  ALWAYS a file -- never a built-in name.
  -n, --none         empty backplane: no boards, no memory, nothing.
  -l, --list         list the built-in machines and exit.
  -s, --script <f>   run a command script, then exit with its status. Paths in
                     it are relative to the script's folder.
  -x, --exec <cmd>   run one monitor command (repeatable), then exit.
  -i, --interactive  after --script/--exec, stay in the monitor.
      --set <text>   change a setting for this run, before the machine's startup
                     commands (repeatable). The text is what follows SET at the
                     prompt: --set "cpu0 clock_hz=2000000".
                     -s, -x, -i and --set cannot be used with --mcp.
      --mcp          MCP server on stdio.
      --mirror <sink>  with --mcp: mirror the console so a person can watch and
                     take over. socket:PORT (telnet in), or on macOS and Linux
                     pty or pty:LINK (open the link with a terminal program).
                     Add ?ro for watch-only.
      --log <file>   with --mcp: write what the guest prints to a text file, as
                     it prints it. An ordinary file; starts empty each run.
  -v, --version      print the version and exit.
  -h, --help         print this help and exit.
```

Monitor commands

Type the part before the bracket.

Command	Does	Usage
`BO[ARDS]`	List, add, or remove boards on the backplane.	`BOARDS [LIST]|ADD <type> <id> [k=v...]|REMOVE <id>`
`B[REAK]`	Set a breakpoint on an address, memory/I/O access, or tape stop.	`BREAK [<addr> | MEM R|W <addr> | IO R|W <port> | TAPE STOP] [IF <expr> | LOADS <expr>] [TRACE ON|OFF]`
`COM[PARE]`	Compare a range of memory against another address.	`COMPARE <range> <addr>`
`C[ONFIG]`	Load or save the whole machine as a TOML file.	`CONFIG LOAD <f.toml> | CONFIG SAVE <f.toml>`
`CONN[ECT]`	Attach a serial unit to an endpoint (console, socket, file, ...).	`CONNECT <id>:<u> <endpoint>`
`CONS[OLE]`	Shortcut for SHOW CONSOLE and SET CONSOLE.	`CONSOLE [<k>=<v>...]`
`DE[POSIT]`	Write bytes into memory at an address.	`DEPOSIT <byte> | DEPOSIT NEXT <bytes...> | DEPOSIT <addr> <bytes...>`
`DI[SASM]`	Disassemble memory into instructions.	`DISASM [<addr>|<range>] [n] [CPU=8080]`
`DISC[ONNECT]`	Shortcut for CONNECT : null: unplug the endpoint.	`DISCONNECT <id>:<u>`
`DO`	Run a file of monitor commands, one per line, as if typed.	`DO <file>`
`D[UMP]`	Show memory as hex and ASCII.	`DUMP [<addr>|<range>] [WIDTH=16]`
`E[DIT]`	Enter bytes into memory interactively from an address.	`EDIT <addr> [ROM]`
`EX[AMINE]`	Point the front panel at an address (and show that byte).	`EXAMINE [<addr>]`
`F[ILL]`	Fill a range of memory with a byte.	`FILL <range> <byte>`
`HE[LP]`	Show help for a command.	`HELP [<command>]`
`H[ISTORY]`	Replay the recent instruction (or bus-cycle) history.	`HISTORY [BUS|CPU] [n]`
`I[N]`	Read a byte from an I/O port.	`IN <port>`

`L[OAD]`	Load a file into memory (binary, Intel hex or S-record).	`LOAD <file> [AT <addr>] [FORMAT=BIN|HEX|SREC] [ROM]`
`MA[CHINE]`	Load a built-in machine by name (MACHINE none empties the backplane).	`MACHINE <name> | MACHINE none`
`M[OUNT]`	Put a disk or tape image into a drive; a .imd is converted to a raw .dsk beside it.	`MOUNT <id>[:<u>] <file>|tnfs://<host>[:<port>]/<path> [WP] [CREATE] [extract[=<base>]] [k=v...]`
`MOV[E]`	Copy a range of memory to another address.	`MOVE <range> <dest> [ROM]`
`N[EXT]`	Step one instruction, running any CALL/RST to completion.	`NEXT`
`NO[BREAK]`	Remove a breakpoint, or all of them.	`NOBREAK [id]`
`NOP[ASTE]`	Stop a PASTE that is not finished.	`NOPASTE`
`O[UT]`	Write a byte to an I/O port.	`OUT <port> <byte>`
`PA[STE]`	Send a host file of any size to the guest's keyboard, as if pasted.	`PASTE <file>`
`P[OWER]`	Power-cycle the machine -- the only thing that clears RAM.	`POWER`
`Q[UIT]`	Leave the simulator.	`QUIT`
`REGI[ON]`	Add a RAM or ROM region to a memory board.	`REGION ADD <id> type=ram|rom at=<addr> [size=|mount=]`
`RE[GS]`	Show the CPU registers (SET REG changes one).	`REGS | SET REG <r>=<v>`
`RES[ET]`	Reset the machine, keeping RAM (RESET CPU resets just the processor).	`RESET [CPU]`
`REST[ORE]`	Load machine state back from a snapshot.	`RESTORE <file>`
`R[UN]`	Start or resume the machine, optionally at an address.	`RUN [addr]`
`SA[VE]`	Write a range of memory out to a file.	`SAVE <file> <range> [FORMAT=BIN|HEX|OCTAL|PRN]`
`SEA[RCH]`	Find bytes or a string in a range of memory.	`SEARCH <range> <bytes...>|"str"`

`SE[T]`	Change a property of a board, the console, display, a register, the bus, or the machine.	`SET <id>[:<u>]|CONSOLE|DISPLAY|TERMINAL|MACHINE|REG|BUS <k>=<v>`
`SH[OW]`	Display the state of a board, the bus, or the machine.	`SHOW <id>|BOARDS|BOARD <type> [UNITS]|MACHINES|MACHINE [<name>]|BUS [MAP|IO|IRQ|CONTENTION]|ROMS|MOUNTS|PATHS|CONSOLE|DISPLAY|TERMINAL|JOYSTICKS|SERIAL|DEBUG|SYM`
`SN[APSHOT]`	Save the whole machine state to a file.	`SNAPSHOT <file>`
`STA[RTUP]`	Edit the machine's boot list (the commands CONFIG SAVE saves as startup = [...]).	`STARTUP [ADD <command> | REMOVE <n> | CLEAR]`
`S[TEP]`	Run one instruction (or n), showing the registers after each.	`STEP [n]`
`SY[MBOLS]`	Load or clear a symbol table for disassembly.	`SYMBOLS LOAD <file> [REPLACE] | SYMBOLS CLEAR`
`T[RACE]`	Log every bus cycle while the machine runs.	`TRACE ON|OFF [file] [MASK=IN,OUT,IRQ,DMA,CONTENTION]`

`TY[PE]`	Feed text to the guest as if typed at its keyboard.	`TYPE "text"`
`U[NMOUNT]`	Take a disk or tape out of a drive.	`UNMOUNT <id>:<u>`
`W[HO]`	Say which board answers an address or I/O port.	`WHO <addr> | WHO IO <port>`

Boards

CPU

Type	What it is
`8080`	MITS 88-CPU: 8080A CPU board
`8085`	Generic 8085 CPU board
`z80`	Generic Z80 CPU board

Memory

Type	What it is
`bankmem`	Bank-switched RAM (Vector, Cromemco, North Star, ExpandoRAM)
`memory`	RAM/ROM board: plain, unbanked memory regions
`v2z80rom`	S100Computers V2 Z80: paged monitor EEPROM

Disk

Type	What it is
`16fdc`	Cromemco 16FDC: floppy controller + console UART, RDOS 2.52
`64fdc`	Cromemco 64FDC: floppy controller + console UART, RDOS 3.12
`dcdd`	MITS 88-DCDD: 8" hard-sector floppy controller
`disk1`	CompuPro Disk 1: uPD765A floppy controller with DMA
`disk1a`	CompuPro Disk 1A: uPD765A floppy controller with DMA
`dualide`	S100Computers IDE-AB: two CompactFlash sockets for CP/M 3
`dualsd`	S100Computers Dual SD: two microSD sockets for CP/M 3
`fdcplus`	FarmTek FDC+: serial drive (drive types 6, 7)
`hdsk`	MITS 88-HDSK Datakeeper: Pertec hard disk controller
`icom`	iCOM FD3712/FD3812: 8" floppy controller with boot PROM
`mds`	MITS 88-MDS: 5.25" minidisk controller
`mdsa`	North Star MDS-A: single-density 5.25" floppy controller
`mdsad`	North Star MDS-A-D: double-density 5.25" floppy controller
`tarbell`	Tarbell #1011: single-density floppy controller
`tarbelldd`	Tarbell #2022: double-density floppy controller
`versafloppy`	SD Systems VersaFloppy I/II: WD177x floppy controller

Serial

Type	What it is
`2sio`	MITS 88-2SIO: two 6850 serial ports
`gsio`	Generic SIO: two strap-configurable serial channels
`io4`	SSM IO-4: two serial + two parallel ports
`pmmi`	PMMI MM-103: Bell 103 modem
`propio`	S100Computers Console IO: Propeller console serial port
`sbc`	SD Systems SBC-100/200: Z80 single-board computer
`sio`	MITS 88-SIO: one serial port (COM2502 UART)
`turnkey`	MITS 8800b Turnkey Module: boot PROM, serial, auto-start

Tape

Type	What it is
`acr`	MITS 88-ACR: audio cassette interface
`uio`	MITS 88-UIO: serial port + cassette interface

Parallel and printer

Type	What it is
`4pio`	MITS 88-4PIO: up to four 6820 parallel ports
`c700`	MITS 88-C700: Centronics line-printer controller
`d7a`	Cromemco D+7A: analog + parallel I/O, joysticks
`lpc`	MITS 88-LPC: 88-LP line-printer controller
`music6`	Newtech Model 6 Music Board: 6-bit D/A and speaker
`pio`	MITS 88-PIO: 8-bit parallel port

Video

Type	What it is
`cadzilla`	CADzilla: HD63484 ACRTC graphics board with a Bt453 RAMDAC
`dazzler`	Cromemco Dazzler: color graphics
`dazzler2`	Cromemco Dazzler II: color graphics, on-board RAM
`vdb8024`	SD Systems VDB-8024: 80x24 video terminal board
`vdm1`	Processor Technology VDM-1: 16x64 memory-mapped video

Systems

Type	What it is
`sol`	Processor Technology Sol-PC I/O: serial, keyboard, tape

PROM programmer

Type	What it is
`pb1`	SSM PB1: 2708/2716 EPROM programmer

Other

Type	What it is
`fp`	Altair front panel: the SENSE switches at IN 0FFH
`hostbridge`	Host Bridge: file transfer to the host (not a period board)
`rtc100`	SciTronics RTC-100: battery-backed clock/calendar
`ss1`	CompuPro System Support 1: interrupts, timer, clock, serial
`virtc`	MITS 88-VI/RTC: vectored interrupts + real-time clock

Machines

Machine	What it is
`8085`	A minimal 8085 machine: an `8085` CPU, 64K of RAM, and a 2SIO console.
`acuter`	ACUTER at F000 -- CUTER on a plain Altair, with a terminal instead of a VDM.
`altmon`	An Altair with a monitor in ROM and a terminal on it.
`amon`	AMON 3.1 in a 4K EPROM at F000 -- Martin Eberhard's full-featured Altair monitor.
`bankmem`	A bank-switched RAM machine: a Z80, a console, and a Vector Graphic 64K bankmem.
`basic4k`	The machine Altair 4K BASIC was sold to run on: an 88-SIO Teletype, a cassette in the ACR.
`basic8k`	The machine Altair 8K BASIC was sold to run on: an 88-2SIO terminal, a cassette in the ACR.
`cadzilla`	An HD63484 ACRTC graphics board with a Bt453 RAMDAC in an Altair -- the bench for CADzilla.
`cdbl`	The `default` machine with the Combo Disk Boot Loader in the PROM socket.
`compupro`	A stock Altair with a CompuPro System Support 1 board for its clock/calendar.
`cuter`	CUTER 1.3 driving a Processor Technology VDM-1 -- the real Sol/CUTS monitor.
`dazzler`	A Cromemco Dazzler in an Altair -- the bench for the S-100's first color graphics card.
`default`	The machine you get when you name none: 56K, and the DBL boot PROM at FF00.
`disk1`	CompuPro Disk 1 floppy: the controller at C0, its boot EPROM and an Interfacer I console. You supply the disk.
`disk1a`	CompuPro Disk 1A floppy: the controller at C0, its boot EPROM and a System Support 1 console. You supply the disk.
`dualide`	S100Computers "IDE-AB CF" machine -- a V2 Z80 CPU board booting CP/M 3 off a CompactFlash card.
`dualidesd`	S100Computers "IDE-AB CF+ESP32" machine -- both boards, CP/M 3 on CF drives A:/B: and SD drives C:/D:.
`dualsd`	S100Computers "Dual SD" machine -- a V2 Z80 CPU board booting CP/M 3 off a microSD card.
`icom`	iCOM FD3712 8" floppy machine -- boots CP/M 2.2 off a single-density iCOM disk.
`lineprinter-lpc`	The `default` machine with an 88-LPC line printer at port 02; CONNECT `lpt0:prn` to a file or the console.
`lineprinter`	The `default` machine with an 88-C700 line printer at port 02; CONNECT `lpt0:prn` to a file or the console.
`minidisk`	The Altair Minidisk: an 88-MDS at 08 and the MDBL boot PROM. You supply the 5.25" disk.
`northstar`	North Star MDS-A single-density floppy: the controller at E800 and its boot PROM. You supply the disk.

`northstardd`	North Star MDS-A-D double-density floppy: the controller at E800 and its boot PROM. You supply the disk.
`original`	The Altair as it actually left Albuquerque.
`parallel`	The `default` machine with two MITS parallel boards: an 88-PIO and an 88-4PIO.
`ps2`	The machine MITS Programming System II ran on: basic8k's cards, but not basic8k's bootstrap.
`ps2int`	MITS Programming System II, WITH INTERRUPTS. `ps2` with A9 down and an 88-VI/RTC in it.
`rombasic`	MITS Extended ROM BASIC 16K -- Extended BASIC that runs directly out of ROM.
`sbc200`	SD Systems SBC-200 -- a 4 MHz Z80 single-board computer running the MSMONR21 monitor.
`sbc200v`	The SD Systems SBC-200 with a VDB-8024 VIDEO console: SDMONV21 on an 80x24 screen.
`sol20`	The Processor Technology Sol-20 -- an integrated 8080 machine, running SOLOS.
`tarbell`	Tarbell #1011 single-density floppy machine -- boots CP/M 2.2 the moment a disk is in it.
`tarbelldd`	Tarbell #2022 double-density floppy machine -- boots CP/M 2.2 the moment a disk is in it.
`turnkey`	The MITS 8800bt -- an Altair with a Turnkey Module where the front panel used to be.
`vdm1`	A Processor Technology VDM-1 in an Altair, and a demo that draws on it.
`z80`	A minimal Z80 machine: a `z80` CPU, 64K of RAM, and a 2SIO console.

A machine file, in one look

```
[machine]
name    = "mine"
base    = "default"        # start from a machine, and say what is DIFFERENT
startup = ["RUN FF00"]     # the operator's own keystrokes. There is no BOOT verb.

[[board]]                  # type + a NEW id      -> ADD the card
type = "2sio"              # type + an id from the base -> REPLACE it outright
id   = "sio0"              # NO type + an id      -> MODIFY the one already there
port = 10                  # remove = true        -> PULL THE CARD OUT

  [board.unit.a]           # a unit's own settings
  connect = "console"

  [[board.region]]         # a list the card owns (memory)
  type = "ram"
  at   = 0000              # hex: it is an address
  size = "56K"             # decimal: it is a size

  [[board.drive]]          # a list the card owns (disk controllers)
  unit  = 0
  mount = "cpm.dsk"        # relative to THIS FILE

[console]                  # the window you started altairsim in -- not a board
strip7out = true
base      = octal          # read/print the wire class in split octal (MITS style)
```

Paths: a path *inside* a machine file is relative to **that file**. A path you *type* is relative to **your shell**.

Endpoints — `CONNECT <id>:<unit> <endpoint>`

Endpoint	Is
`console`	the host terminal. Exactly one unit may hold it.
`null`	nowhere. Writes vanish, reads never come.
`loopback`	itself — what you write comes back.
`scripted`	a caller in place of a human — what MCP and the tests type into.
`socket:PORT`	**listens**, as a raw pipe — for a program at the far end. `?banner` greets each caller.
`socket:HOST:PORT`	**calls out**, as a raw pipe.
`telnet:PORT`	**listens**, speaking Telnet — this is telnet-in for a person. Greets each caller; `?banner=off` stops it.
`telnet:HOST:PORT`	**calls out**, taking the telnet client's part.
`serial:DEVICE`	a real serial port on this host.
`pty`	a pseudo-terminal (macOS and Linux): open `/tmp/altairsim0` with a terminal program. `pty:LINK` selects the name.
`in:PATH`	a host file as a reader (paper tape). `?cps=N` paces it.
`out:PATH`	a host file as a punch — 8-bit clean, never truncating.
`terminal`	a window the simulator draws itself (SDL builds). `?emulation=vt100|adm3a|vt52|h19`, `?size=COLSxROWS`.
`printer:QUEUE`	a real print queue on this host.
`<endpoint>|FILE`	a tap: append `|FILE` to any endpoint to also log the line. `?fmt=text` = plain text, only what the guest prints.
`<endpoint>|socket:PORT`	a live mirror: `telnet` in to watch and take over. `?ro` = watch-only.
`<endpoint>|pty[:LINK]`	the same mirror on a pseudo-terminal (macOS and Linux): open `/tmp/altairsim0`, or `LINK`, with a terminal program.
`<endpoint>|serial:DEVICE`	the same mirror on a real serial port, 8N1: a hardware terminal watches and types. `?baud=N` (default 9600), `?ro`.

Machines

A **machine** is a backplane with boards in it. The boards, their settings and their state are the whole machine. That is all that `altairsim` needs to know.

You choose the machine in one of three ways. You give the name of a **built-in** machine, you give a **file**, or you give **nothing** and get the default. This chapter tells you how the program makes that choice, and what a path means.

The command line

```
altairsim [machine] [options]
```

Give the machine first, and then the options for that machine. The program also accepts the options before the machine: `altairsim mine.toml -s check.cmd` and `altairsim -s check.cmd mine.toml` do the same thing.

Option	
`machine`	a built-in name, **or** a machine file if it contains a `/` or ends in `.toml`
`-m, --machine <name>`	**always** a built-in name, never a file
`-f, --file <path>`	**always** a file, never a built-in name
`-n, --none`	an empty backplane. No boards, no memory, nothing
`-l, --list`	list the built-in machines and exit
`-s, --script <file>`	run a command script, then exit with its status. Paths in it are relative to the script's folder
`-x, --exec <cmd>`	run one monitor command, then exit. You can give it more than one time
`-i, --interactive`	after `--script` or `--exec`, stay in the monitor
`--set <setting>`	change a setting for this run, before the machine's startup commands. You can give it more than one time
`--mcp`	run as an MCP server on stdio. You cannot give `-s`, `-x`, `-i` or `--set` with it
`--mirror <sock>`	with `--mcp`, copy the console to a socket so that a person can watch
`-v, --version`	print the version and exit
`-h, --help`	print this help and exit

Give one machine only. If you give a name *and* `-m`, or `-f` *and* `-n`, the program prints *give ONE machine* and stops. It does not choose one for you.

How a bare word resolves

```
$ altairsim basic4k                    a BUILT-IN, by name
$ altairsim examples/cpm/cpm22-buffered.toml            a FILE, by path
```

The program decides from the word only. **It never looks for a file on your disk:**

The word	What it is
contains `/` or `\`	a **file**
ends in `.toml`	a **file**
anything else	a **built-in name**

Because of this rule, **`altairsim basic4k` means the same machine in every folder.** A file called `basic4k` in your folder does not change it. A command in a script or in a README always means the same thing.

For a file, you must give the `.toml` extension, as in `altairsim mymachine.toml`, or a `/`, as in `altairsim ./mymachine`. To say which one you mean, use `-m` or `-f`:

```
$ altairsim -m basic4k                 a built-in. Stop looking for a file
$ altairsim -f ./basic4k               a file called `basic4k`, no extension, right here
```

The one file the simulator *finds*

If you give **nothing**, and the working folder has a file called `altairsim.toml`, the program loads that machine. It tells you that it did:

```
$ altairsim
altairsim: no machine named -- using ./altairsim.toml (`-m default` for the built-in).
AltairSim X.Y.Z -- 8080, full speed.
machine: bench.  HELP for commands.
altairsim>
```

The first line tells you which file the program loaded. The program prints it before anything else. The `machine:` line gives the name that is written *in* the file, `bench` here.

This is the **only** file that the program finds by itself. It does this only when the command line gives no machine. If you give a built-in name, a file or `-n`, the program ignores `./altairsim.toml`.

Put an `altairsim.toml` in a project folder, and `altairsim` with no arguments loads your machine. With no `altairsim.toml` and no arguments, you get the built-in `default` machine.

The built-in machines

A **built-in machine is a TOML machine file that is stored in the program.** It uses the same format, keys and rules as a machine file that you write. It is in the program so that it is always there.

```
$ altairsim --list
```

This command lists the built-in machines, one on each line, with a sentence about each. The list comes from the program, so it always has every machine. The machine reference at the back of this manual has the same table. At the `altairsim>` prompt, `SHOW MACHINES` gives the list.

To see what is in a built-in machine, its boards and its startup commands, give its name:

```
$ altairsim basic4k -x 'SHOW MACHINE'
```

At the prompt, type `SHOW MACHINE basic4k`. `SHOW MACHINE` with no name shows the machine that you are running now.

To get the machine file as **text that you can edit**, with every board and every setting:

```
$ altairsim basic4k -x 'CONFIG SAVE mine.toml'
$ altairsim mine.toml
```

`CONFIG SAVE` saves the machine that you are running, and the file loads back as the same machine. **You can use each built-in machine as an example.** Find the one that is nearest to what you want, save it, and edit the file.

You can also start *from* a built-in machine with `base`, and write only what is different. The configuring chapter tells you how.

How the machine is wired: `SHOW BUS`

`SHOW MACHINE` tells you which boards are in the machine. `SHOW BUS` tells you how they are wired together. It shows what answers at each memory address, what answers at each port, and what happens

on an interrupt. Use it first on a machine that you built. For example, type `altairsim basic4k -x 'SHOW BUS'`, or type `SHOW BUS` at the prompt:

```
altairsim> SHOW BUS
MEMORY
  0000-0FFF  mem0     ram
  unmapped: 1000-FFFF  (floats to FF)
I/O
  FF         fp0      IN  --     SENSE switches SA8..SA15 -> D0..D7
  00         sio0     IN  OUT    COM2502 -- status / interrupt enables
  01         sio0     IN  OUT    COM2502 -- data
  06         acr0     IN  OUT    88-SIO B UART -- status / interrupt enables
  07         acr0     IN  OUT    88-SIO B UART -- data, via the modem, to the cassette
INTERRUPTS
  CPU     INTE off         interrupts are DISABLED; nothing will be acknowledged
  pINT    idle             pin 73
  88-VI   (none)           nothing watches VI0-VI7; an acknowledged interrupt
                           floats to FF, which the 8080 executes as RST 7
```

The output has three sections:

- **`MEMORY`** lists each range and the board that answers it. A range with no board is `unmapped`. This is not an error: a read from it returns `FF`, the same as an empty socket on a real backplane. A boot PROM that the program has switched out is not in the list.
- **`I/O`** lists each port and the board that answers it. The `IN` and `OUT` columns show which direction the board answers. `--` means that the board does not answer that direction. A port that is not in the list returns `FF` on an `IN`.
- **`INTERRUPTS`** is the short form of `SHOW BUS IRQ`. It shows whether the processor accepts interrupts, whether the interrupt pin is asserted, and whether an 88-VI board is there to read lines VI0 to VI7. It lists each line that has a board set to it. A board that is set to a line that nothing reads does not give an error. The machine runs, and it never gets the interrupt. A `WARNINGS` part shows you this fault.

To see only one section, or more detail, add a view: `SHOW BUS MAP`, `SHOW BUS IO` or `SHOW BUS IRQ`. `SHOW BUS CONTENTION` shows two boards that answer the same address or port. The boards chapter has an example. The Monitor document has the full list of views.

The empty backplane: `-n`

```
$ altairsim -n
```

`-n` gives a machine with no boards, no memory and no processor. You add every board yourself with `BOARDS ADD`. Use it when you build a machine one board at a time. It is also the only way to be sure that

the machine has nothing in it that you did not add.

From an empty backplane to a machine file

These steps build a machine at the prompt, save it to a file, and load it again:

```
$ altairsim -n
altairsim> BOARDS ADD 8080 cpu0
cpu0: 8080 added
altairsim> BOARDS ADD 2sio sio0 port=10
sio0: 2sio added
altairsim> SET sio0:a connect=console
sio0:a: connect=console
altairsim> BOARDS ADD memory mem0 fill=random
mem0: memory added
altairsim> REGION ADD mem0 type=ram at=0 size=56K
mem0:0: ram  0000-DFFF  56K
altairsim> REGION ADD mem0 type=rom at=FF00 mount=builtin:dbl
mem0:1: rom  FF00-FFFF  builtin:dbl
altairsim> CONFIG SAVE mine.toml
saved mine.toml
altairsim> QUIT
```

To get the machine back, start with an empty backplane again and load the file:

```
$ altairsim -n
altairsim> CONFIG LOAD mine.toml
loaded mine.toml: 3 board(s)
```

`$ altairsim mine.toml` does the same thing in one step.

Three things in these steps are important:

- **You can give a board's settings on the `BOARDS ADD` line** (`port=10`), or set them later with `SET`. The result is the same.
- **`SET` sets one property in each command**, and the board must exist first.
- **A memory board has no memory until you add regions to it.** Use `REGION ADD`. Do not forget the ROM region, because it holds the code that `RUN` starts.

Two more commands are often part of a machine. `CONFIG SAVE` saves both of them:

- **`STARTUP ADD RUN FF00`** records the command that starts the machine. Use `MOUNT` to put a disk in a drive.
- **`SET MACHINE name=<name>`** gives the machine a name. A machine that you build from `-n` is called `none` until you give it a name. `CONFIG SAVE` saves the name. The name is a label: the `base =` key of another file finds this machine by the path of its file, not by its name.

The path rule: one base directory

This rule lets you copy an example folder to any place, and the machine still boots:

> **A relative path resolves against the machine's directory**, which is the folder that the machine file was loaded from.

That folder is the base for the disks and PROMs that the machine file mounts. It is also the base for the `MOUNT`, `LOAD`, `SAVE` and `DO` paths that you type. The rule is the same for a path in the file and a path that you type.

A **script** follows the same idea. A path in a script is relative to the script's folder, as a path in a machine file is relative to the machine file. This is true for a script that you run with `DO` and for a script that you run with `-s`.

`examples/cpm/cpm22-buffered.toml` has `mount = "cpm22b23-56k.dsk"`. This means *the disk in this folder*. It still means that after you copy the folder, rename it or send it to another person. For this reason, each example is a folder that has all its files, and the `cp -R` in the quick start works. If you then type this:

```
altairsim> MOUNT dsk0:drive1 cpm22b23-56k.dsk
```

you get the **same file**, from the **same folder**, the folder of the machine file. The folder that you started `altairsim` from does not change this.

A **built-in** machine has no folder of its own. Its base is the folder that you started the program from.

When the rule causes a problem

You see the rule only when a file is missing. When that happens, the error can look like a typing mistake. For example, you keep your machine files in a `machines/` folder. You write a path that starts from the folder that you start the program from:

```toml
[[board.drive]]
unit  = 0
mount = "disks/Kermit/cpm.dsk"      # meant: the disks/ up beside machines/
```

`altairsim -f ./machines/8800c.toml` then prints the error, with the line of the `[[board.drive]]` in your file:

```
./machines/8800c.toml: line 12: dsk0: 'machines/disks/Kermit/cpm.dsk': no such file
  ('disks/Kermit/cpm.dsk' is relative to the machine's directory, ./machines/)
```

The disk is not missing. The program looked for it in the folder of the machine file, because that is the machine's directory. Write the path from that folder:

```
mount = "../disks/Kermit/cpm.dsk"   # up out of machines/, then down into disks/
```

You can also keep the machine file in the same folder as its disks, as every example in the package does. The same `../` applies to a path that you type, because both start from the same base.

Ask the machine

`SHOW PATHS` prints the base for the machine that you are running:

```
altairsim> SHOW PATHS
  base directory     /home/you/altair/disks/cpm22
                     What a machine file mounts, and the MOUNT / LOAD / SAVE /
                     DO you type, resolve against this. A path inside a DO or
                     -s file is relative to that file.
                     It is the directory the machine was loaded from.

  hb0 sandbox        /home/you/altair/disks/cpm22/xfer
                     THE GUEST'S SANDBOX, and the only real fence here:
                     R.COM/W.COM cannot leave it. It is not a base for
                     anything you type. Set with `hostdir`.
```

The two entries do different jobs. The base directory is where relative paths start. The sandbox is the only folder that the guest can read and write.

For a **built-in** machine, the base is the folder that you started the program from:

```
  base directory     /home/you/altair
                     ...
                     This machine is built in, so it is the directory you
                     launched from.
```

`SHOW MOUNTS` lists every disk, tape and ROM in the machine, for all the boards:

```
altairsim> SHOW MOUNTS
  UNIT         KIND  HOLDS
  dsk0:drive0  disk  cpm22b23-56k.dsk
  dsk0:drive1  disk  (empty)
  dsk0:drive2  disk  (empty)
  dsk0:drive3  disk  (empty)
  mem0:rom0    rom   builtin:dbl  (read-only)

  Paths are AS WRITTEN.  SHOW PATHS says what they are relative to.
```

Empty drives are in the list. The 88-DCDD has four drives. This machine has a disk in one of them.

`SHOW MOUNTS` shows each path as it was written. `SHOW PATHS` tells you which folder those paths are relative to. Use the two commands together.

The path rule is not a sandbox

The path rule tells the program **where a path points**. It does not limit anything. A machine file can mount any file on your disk, with `..` or with an absolute path.

The only limit is the **`hostdir`** of the Host Bridge. It limits the files that a CP/M program *in* the machine can read and write on your computer. This is a different mechanism for a different purpose. The chapter *Moving files in and out* describes it. Nothing in a machine file's paths changes it.

Changing settings for one run: `--set`

A machine file gives each setting one value. To run the machine with a different value, give the setting on the command line with `--set`. The machine file does not change, and you do not need a second machine file.

For example, `cpm22-buffered.toml` runs CP/M at full speed. This command runs the same machine at 2 MHz:

```
$ cd examples/cpm
$ altairsim cpm22-buffered.toml --set "cpu0 clock_hz=2000000"
AltairSim X.Y.Z -- 8080, full speed.
machine: cpm22-buffered.  HELP for commands.
set> SET cpu0 clock_hz=2000000
cpu0: clock_hz=2000000
startup> RUN FF00
[console -- ^E returns to the monitor]

56K CP/M 2.2b v2.3
For Altair 8" Floppy

A>
```

The first two lines show the machine that the file made, which runs at full speed. The `set>` line shows the change. Without `--set`, the machine runs at full speed again:

```
$ cd examples/cpm
$ altairsim cpm22-buffered.toml
```

The text after `--set` is what you type after `SET` at the monitor. `SET cpu0 clock_hz=2000000` at the monitor becomes `--set "cpu0 clock_hz=2000000"` on the command line. The text has a space in it, so put it in quotation marks. The keys are the keys of the machine file. `SHOW cpu0` lists the keys of a board, and the boards chapter describes them.

The program applies each setting before the machine's startup commands. The order is:

1. The program loads the machine file, and shows what it loaded.
2. It runs `~/altairsim.ini`, if you have one. A later section describes this file.
3. It applies each `--set`, in the order that you gave them, and shows each one behind `set>`.
4. It runs the `startup` commands of the machine file.

A `--set` comes after your `~/altairsim.ini`. If the two set the same key, the `--set` is the value that the machine has.

This order is why `--set` works for a machine that starts by itself. `cpm22-buffered.toml` has `startup = ["RUN FF00"]`, so CP/M starts as soon as the file is loaded. A `-x "SET ..."` runs after the startup commands, when CP/M has started. `--set` runs before them.

You can give `--set` more than one time. This command puts the CP/M console in a terminal window of its own:

```
$ cd examples/cpm
$ altairsim cpm22-buffered.toml --set "sio0:a connect=terminal" --set "TERMINAL bsdel=bs"
```

`sio0:a` is unit `a` of the serial board, which is the console of this machine. `connect` is the device on that line, and `terminal` is a VT100 in a window. `TERMINAL bsdel=bs` makes the Backspace key in the window send the byte that CP/M expects. CP/M starts in the window, and the monitor stays in the terminal where you started the program. The serial chapter describes `terminal` and its options.

To run that machine at 2 MHz also, add `--set "cpu0 clock_hz=2000000"` to the same command.

- **The machine file does not change.** `--set` is for this run only.
- **A setting that fails stops the program.** If the board, the key or the value is wrong, the program shows the error from `SET`, then `--set failed; the machine was not started.` The exit status is 2. The startup commands do not run.
- **You can use `--set` with `-x`, `-s` and `-i`.** The program applies the settings before it runs the commands.
- **You cannot use `--set` with `--mcp`.** An MCP client sends `SET` with the `monitor` tool.

Running a command and leaving: `-x` and `-s`

You can use `altairsim` without typing at the prompt.

```
$ altairsim default -x 'SHOW MACHINE'
$ altairsim examples/cpm/cpm22-buffered.toml -x 'DUMP 0 F'
```

`-x` runs one monitor command on the machine, and then the program exits. You can give `-x` more than one time. The commands run in the order that you give them:

```
$ altairsim examples/cpm/cpm22-buffered.toml -x 'MOUNT dsk0:drive0 mine.dsk' -x 'RUN FF00' -i
```

Without `-i`, the program exits when the commands are done. With `-i`, you get the monitor, and the machine is as your commands left it. `-i` with no `-x` or `-s` has no effect.

`-s` runs a **script**. A script is a file of monitor commands, one on each line. These are the same commands that you type at the prompt. Most example folders have a script, with the extension `.ini`. For example, `cpm22-buffered.ini` builds the same CP/M machine as `cpm22-buffered.toml`, and boots it:

```
$ cd examples/cpm
$ altairsim -s cpm22-buffered.ini
```

At the `altairsim>` prompt, `DO cpm22-buffered.ini` runs the same script.

A path in a script is relative to the script's folder. The script above mounts `cpm22b23-56k.dsk`, the disk beside it. So the script also works from another folder. Give the path to the script from the folder that you are in:

```
$ altairsim -s examples/cpm/cpm22-buffered.ini
```

The exit status is not zero if a command failed. You can use `altairsim -s` in a shell script, a Makefile or a build, and test the result:

```
if altairsim mine.toml -s check.cmd; then
    echo "machine is sane"
fi
```

This command has one script and one machine. The program loads the machine `mine.toml`, and runs its startup commands. It then runs each line of `check.cmd` on that machine. A script takes no arguments.

A script that runs at every start: `~/altairsim.ini`

If there is a file called `altairsim.ini` in your home folder, `altairsim` runs it each time that you start the monitor. It is a script, the same kind as `-s` runs: monitor commands, one on each line. Use it for settings that belong to you and not to a machine, for example where the debug output goes. This line writes it to a file in your home folder:

```
# ~/altairsim.ini
SET CONSOLE DEBUG=altairsim-debug.log
```

The file runs after the machine is loaded and before the machine's own `startup` commands, so a machine that starts running at once still gets your settings. A `--set` on the command line is applied after the file runs, and before the `startup` commands. The program shows each line behind `ini>` as it runs it.

- **No file is not an error.** Without the file, nothing happens.

- **A command that fails does not stop the file.** The program shows the error, runs the next line, and the exit status is not changed.
- **A path in the file is relative to your home folder.**
- **`--mcp`, `-x` and `-s` do not run it.** Their output is read by a program, and a file in your home folder must not change it. To use the same lines in a script, name the file: `altairsim -s ~/altairsim.ini`.

Which chapter next

The **configuring** chapter describes the machine file: every table, every key, and the four forms of a `[[board]]` entry. The **boards** chapter describes each board.

Configuring a machine

A machine file is a **TOML** file that describes a machine. It lists the boards in the backplane, the settings of each board, and the commands to run after the power comes on. This chapter describes the format in full.

What TOML is

TOML is a plain configuration format. You can write it in an editor, or the `CONFIG SAVE` command can write it from the machine that you are running. Either way, you can read it again later without a manual. These rules are enough to read every example in this chapter:

- `key = value` gives one setting on each line, for example `clock_hz = 2000000`.
- **Quotes are for text.** A string has quotes (`name = "cpm22"`). A number, `true` or `false` has no quotes (`size = 256`, `idle = true`).
- **A `[table]`, with single brackets, is a thing that the machine has one of.** `[machine]` is the machine, and `[console]` is the window you started altairsim in. You write it one time.
- **A `[[table]]`, with double brackets, is a thing that the machine has many of.** Each `[[board]]` starts one more board, and each `[[board.drive]]` one more drive. The double brackets mean "add one more to the list". Write `[[board]]` for each board.
- **A nested table such as `[board.unit.x]` belongs to the block above it**, by its name. The indentation in these examples only makes them easier to read.
- **`#` starts a comment** that continues to the end of the line.

How a machine file is loaded

There are four ways to load a machine file:

- **Name it on the command line**: `altairsim mine.toml`. A filename that has a `/` or ends in `.toml` is a file, not a built-in machine. Use `-f mine` if the filename has neither.
- **Name nothing.** If the working folder has a file called `altairsim.toml`, `altairsim` with no arguments loads it. If it has none, you get the built-in `default` machine.
- **Type `CONFIG LOAD mine.toml`** at the monitor, in a machine that is already running. The file **replaces** that machine, and it loads all of the file or nothing.
- **Name it as the `base` of another file** (`base = "mine.toml"`). The other file starts from this one and gives only what is different.

The first three make the file the machine. The last one uses the file as a start for another. The machines chapter gives the rules for the command line and for relative paths.

The file that you will write

Most machine files start from a machine that already exists, and they give only what is different. This is `examples/cpm/cpm22-buffered.toml` without its comments:

```
[machine]
name = "cpm22-buffered"
base = "default"
startup = ["RUN FF00"]

[console]
bsdel = "bs"

[[board]]
id = "dsk0"

  [[board.drive]]
  unit  = 0
  mount = "cpm22b23-56k.dsk"
```

The `default` machine already has everything that a 56K CP/M machine needs. It has a front panel, an 8080, a 2SIO console, a floppy disk controller, 56K of RAM and the boot PROM at `FF00`. This file adds three things: the disk in drive 0, the `RUN FF00` command, and the Backspace setting that CP/M expects.

The `[[board]]` has no `type`, because `dsk0` is already in the machine, and the file changes it. With `type = "dcdd"`, the file would remove the controller of the base and add a new one with the default settings. The file still loads, with no error, so you do not see the mistake. Leave the `type` out.

Two rules

A machine file can do what you can type, and nothing more. It cannot boot a disk, because there is no `BOOT` command. It can type `RUN FF00` for you, which is what the operator did. Every key on a board is a property that `SET` changes at the monitor, and every property is a key that you can write in the file. The board reference at the back of this manual lists every property of every board. The program prints that reference from the same table that `SET` uses, so it is always current.

An unknown table or key is an error, and the machine does not load. The program tells you which key it did not know, and the line of the file that the key is on:

```
mine.toml: line 4: unknown [machine] key 'widget'. Valid keys: base name startup
mine.toml: line 7: [[board]] cpu0: cpu0 has no property 'frobnicate'. Valid properties: achieved_hz clock_hz
idle
```

For this reason, a setting with a typing mistake never looks as if it worked. You find the mistake when you load the file.

A table or a key that you write two times is an error also. Write `[machine]`, `[console]`, `[display]` and `[terminal]` one time each in a file. Write a `[board.unit.<name>]` one time below its `[[board]]`, and write a key one time in its table. This mistake usually occurs when you paste one machine file into another. The error gives both lines:

```
mine.toml: line 8: [console] is already at line 5 -- a table is written one time
mine.toml: line 6: 'name' is already set at line 2
```

The wrong brackets on a table are an error also. `[board]` with single brackets does not add a board, and `[[console]]` with double brackets is not the console. The error gives the correct form:

```
mine.toml: line 6: [board] must be [[board]] -- double brackets, one for each
mine.toml: line 4: [[console]] must be [console] -- a table that is written one time
```

Each key must be below a table. A key above the first table is an error.

The tables

Table	What it is
`[machine]`	the name of the machine, its base and its startup commands. Three keys only
`[[board]]`	a board. One entry for each board
`[board.unit.<name>]`	one unit *on* the board above it, for example a serial channel or a tape deck
`[[board.region]]`	a memory region on a `memory` board
`[[board.drive]]`	a drive on a disk controller
`[console]`	**the window you started altairsim in.** It is not a board
`[display]`	**the windows that altairsim opens.** It is not a board
`[terminal]`	the built-in terminal window. The serial chapter describes it

`[machine]`: three keys

```toml
[machine]
name    = "cpm22"
base    = "default"
startup = ["RUN FF00"]
```

`name`

The name of the machine. **The name is a label for you, and nothing more.**

- **The program shows it in three places:** the `machine:` line that it prints when the monitor starts, the first line of `SHOW MACHINE`, and the title bar of the video window.
- `SET MACHINE name=` **changes it** at the monitor, and `CONFIG SAVE` saves it.
- **The program does not find a machine file by its name.** The command line and the `base` key find a file by its path. The name in the file can be different from the file name.
- **A file with no `name` has the name of its base.** A file that starts from `default` and gives no name is called `default`.

`base`: start from a machine, and write what is different

```
base = "default"               # a built-in machine
base = "../cpm22/cpm22.toml"   # or a file
```

The program reads the value with **the same rule as the command line**. A value that contains a `/` or ends in `.toml` is a file. Any other value is the name of a built-in machine. The machines chapter gives the rule. A file path is relative to the file that names it.

- **The program reads `base` before every other key**, in any order in the file.
- **`base` must come before the first `[[board]]`.** A file cannot change a backplane and then start from a different one.
- **A base can have its own `base`, up to 8 levels deep.**

With `base`, a machine file contains **only what is different** from its base. You do not copy every board, so a change to the base reaches every file that uses it.

A file that starts from a file

A base can be a machine file that you wrote, and that file can have a base of its own. These two files are in the same folder as the disk image. The first file is a CP/M machine:

```toml
# cpm22.toml
[machine]
name    = "cpm22"
base    = "default"
startup = ["RUN FF00"]

[[board]]
id = "dsk0"

  [[board.drive]]
  unit  = 0
  mount = "cpm22b23-56k.dsk"
```

The second file is the same machine at 2 MHz. It contains the one setting that is different:

```toml
# cpm22-2mhz.toml
[machine]
name = "cpm22-2mhz"
base = "cpm22.toml"

[[board]]
id       = "cpu0"
clock_hz = 2000000
```

- **The chain has three levels:** `cpm22-2mhz.toml`, then `cpm22.toml`, then the built-in machine `default`.
- **The second file gets the disk and the `startup` list from the first.** It does not write them again. A file that has its own `startup` list uses that list and not the list of its base.
- **`base = "cpm22.toml"` is the file in the same folder**, because the path is relative to the file that names it.
- **A change to `cpm22.toml` reaches the two machines.**

If two files name each other as a base, the chain has no end. The program stops at 8 levels and does not load the machine. The message gives the file that you loaded and the line of its `base`:

```
a.toml: line 3: base = "b.toml": more than 8 levels deep -- do two files name each other?
```

`startup`: the operator's commands

```toml
startup = ["RUN FF00"]
```

`startup` is a list of **ordinary monitor commands**. The program runs them in order after it builds the machine, and it shows each one on a `startup>` line.

A path in a `startup` command is relative to the machine file, the same as every other relative path. A `startup` line and the same command that you type find the same file.

What `[machine]` does not accept

```
[machine]
clock_hz = 2000000        # ERROR
sense    = 0x80           # ERROR
```

The program **rejects both, and it tells you where each one goes**:

```
mine.toml: line 4: clock_hz belongs to the CPU BOARD, not to [machine] --
  the crystal is on the board. Put it in the CPU's [[board]]:
      [[board]]
      type     = "8080"
      id       = "cpu0"
      clock_hz = 2000000
```

The crystal is on the processor board, and the sense switches are on the front panel. Neither one is a property of the machine.

`[[board]]`: four forms

What a `[[board]]` entry does depends on two things. Does it have a `type`? Did the base already use its `id`?

Write	What it does
`type` + a **new** `id`	**ADD** the board
`type` + an `id` **from the base**	**REPLACE** the board
no `type` + an `id`	**MODIFY** the board
`remove = true` + an `id`	**REMOVE** the board

Every `[[board]]` needs an `id`. You use the `id` to name the board at the monitor, and a file that uses this one as its base uses the `id` to name the board.

Write `id`, `type`, `remove` and the table names in lower case. TOML is case-sensitive, so `Id` is not `id` and `[[Board]]` is not `[[board]]`. The program refuses a word that has the wrong case, and it shows the right one. For example, `unknown key `Id`; did you mean `id`?` The value of `remove` is `true` or `false`, in lower case. The names of settings and their values are not case-sensitive. A setting that you write two times in one table is an error, also when the two names have different case.

ADD: `type` + a new id

```
[[board]]
type = "virtc"
id   = "vi0"
```

The machine now has this board. **In a file with no `base`, this is the only form**, because there is no board to modify, replace or remove.

REPLACE: `type` + an id that the base already used

```
[[board]]
type = "2sio"
id   = "sio0"
port = 0x20
```

If the base had a board called `sio0`, the program removes it and adds a new `2sio` in its place. **All the settings that the base made on that board are lost**, also the ones that you did not write. You get the defaults of the type, and the settings that you write here.

In a file with a `base`, you almost always want MODIFY. Leave the `type` out.

`CONFIG SAVE` saves every board with its `type` and all its keys. Do not copy a board from a saved file into a file that has a `base`, because that is a REPLACE. Write the `id` and the keys that you change.

MODIFY: no `type`

```
[[board]]
id       = "cpu0"
clock_hz = 2000000
```

When you leave out the `type`, you change the board that is already there. All the settings that the base made on `cpu0` stay. Only the clock changes.

REMOVE: `remove = true`

```
[[board]]
id     = "acr0"
remove = true
```

The program removes the board from the backplane, and its ports are no longer decoded. No other entry in the file can name it.

The same id two times in one file

A `type` with an id that the same file already used is an error. It is not a replace. Two boards with the same id in one file are almost always a block that you copied and did not rename. The file does not load, and the error names the id.

An id from your *base* is different. It is a board that you got from the base, and you can replace it.

Every other key on a `[[board]]` is a property

`type`, `id` and `remove` tell the program which board the entry is. **Every other key goes to the board**, and the board accepts it or rejects it by name:

```
[[board]]
id     = "dsk0"
drives = 8
```

The board reference at the back of this manual lists the keys of every board. The boards chapter tells you what each board is.

`[board.unit.<name>]`: settings for one unit

Some boards have more than one independent part. An 88-2SIO has **two 6850 ACIAs**. Unit `a` and unit `b` each have their own baud rate, their own interrupt setting and their own connection, so each one has its own table:

```
[[board]]
id = "sio0"                    # no type: the board is already in the machine

  [board.unit.a]
  connect = "telnet:2323"      # a property of unit a

  [board.unit.b]
  baud    = 1200               # unit b has its own settings
  connect = "socket:2324"
```

This sample is for a file whose base has the board `sio0`, such as `base = "default"`. It has no `type`, and it gives only the keys that change. The other settings of the board and of each unit stay.

A key in a unit table is a command at the monitor. `connect` in `[board.unit.a]` is `CONNECT sio0:a telnet:2323`, and `baud` in `[board.unit.b]` is `SET sio0:b baud=1200`. The serial chapter shows each `CONNECT` command with its machine-file form. The board reference tells you which boards have units, and the keys of each unit.

`[[board.region]]`: memory

A `memory` board is **a list of regions**. For this reason, one memory board can have 56K of RAM and a boot PROM at the top of memory.

```
[[board]]
type = "memory"
id   = "mem0"

  [[board.region]]
  type = "ram"
  at   = 0x0000            # HEX: it is an address
  size = "56K"             # DECIMAL: it is a count

  [[board.region]]
  type  = "rom"
  at    = 0xFF00
  mount = "turnmon.bin"    # relative to this file
```

Key	
`type`	**required.** `ram` or `rom`
`at`	the address where the region starts. **Hex**
`size`	the size of a `ram` region. **Decimal**. You can use the `K` and `M` suffixes. A `rom` region takes its size from its image, rounded up to a page, so it ignores `size`
`mount`	a ROM image: a file path, or `builtin:<name>`

A size with a suffix needs quotes, as in `size = "56K"`, because TOML does not accept `56K` as a number. A plain number needs no quotes: `size = 256`.

An empty socket

A `rom` region with no `mount` is an empty socket, even when it has a `size`. It decodes nothing, so a read there gets `FF`. That is what an S-100 bus gives when no board drives it. It is not zero, and it is not an error. A real board with an empty PROM socket gives the same result.

`[[board.drive]]` : disks

A disk controller has drives, and a drive holds a disk image.

```
[[board]]
id = "dsk0"

  [[board.drive]]
  unit     = 0             # DECIMAL: it is a drive number
  mount    = "cpm.dsk"     # relative to this file
  readonly = false
```

Key	
`unit`	the drive number. **Decimal**
`mount`	the image file
`readonly`	refuse every write at the controller, so that the file cannot change. The disks chapter tells you more. `writeprotect` is the same key
`media`	the disk format. Use it when the controller cannot detect the format from the image
`create`	make an empty file if the file is not there, and then mount it. This is `MOUNT … CREATE` in the file

The controller usually detects the format from the image. Use `media` when it cannot, for example for an image with no header or with an unusual layout. `media` also sets the size of a **blank** disk, because a blank disk matches no format. The disks chapter describes the formats and `create`.

Without `create`, a `mount` that names a missing file is an **error, and the machine does not load**.

Each `[[board.drive]]` is one `MOUNT` command. The disks chapter shows each `MOUNT` command with its machine-file form.

Numbers: hex on the bus, decimal for counts

The machine file uses the same number rule as the monitor. *The Monitor* gives the full rule. **A value that the processor sees on the bus is hex. A value that never goes on the bus is decimal.**

```
port  = 10        # a port is on the bus, so it is HEX. This is port sixteen.
at    = 0xFF00    # an address: hex
sense = 80        # the sense switches: hex, 0x80
baud  = 9600      # a rate: DECIMAL
size  = 256       # a count: DECIMAL
```

`port = 10` is port sixteen. A port is on the bus, so it is hex, and every listing from 1976 put the 2SIO at 10. To make the base clear in your own files, use a marker:

- `0x10`, `$10` or `10h` for hex
- `0o20` or `20q` for octal
- `%10000` for binary. `0b10000` is binary only in a decimal setting; in a hex setting, `B` is a digit
- `"#16"` for decimal. The quotes are necessary, because `#` starts a TOML comment
- a `K` or `M` suffix, which is always a decimal count

To read and print in octal, set `base = "octal"` in `[console]`. This changes the base, not the rule. `SHOW` prints a port in hex with a `0x` marker, for example `port 0x20`, whatever form you wrote.

`[console]`: the console, which is not a board

```
[console]
stop      = 0x05      # the STOP key, Ctrl-E. attn= also works
base      = "hex"     # hex or octal: how you read and write addresses, ports and bytes
upper     = false
strip7in  = false
strip7out = false
crlf      = false
echo      = false
bell      = true
bsdel     = "bs"
```

`[console]` is **not a `[[board]]`**, and it is not in the backplane. It describes *the console*, the window you started altairsim in. The monitor runs there. The Altair knows nothing about it.

Key	
`stop`	the byte of the STOP key. **Hex.** The default is `05`, which is `Ctrl-E`. `attn` is the same key
`base`	`hex` or `octal`: how the monitor reads and prints addresses, ports and bytes. `octal` is split octal, as on the MITS front panel
`history`	how many lines the monitor's command history file keeps. The default is 50. `0` stops saving it
`log`	a file that gets a copy of the session. `off` stops it
`upper`	change input to upper case
`strip7in`	clear bit 7 of every byte that the guest receives
`strip7out`	clear bit 7 of every byte that the guest sends
`crlf`	change line endings
`echo`	show typed characters locally
`bell`	let the guest ring the bell of the console
`bsdel`	`off`, `bs` or `del`: what your Backspace key sends

These settings are the only part of the program that changes a byte. They belong to the console, because a person reads text there. **Every serial line is 8-bit clean.** No board has a setting that clears bit 7, because a line can carry XMODEM, and a line that clears bit 7 cannot carry a file. For example, to fix the garbled `MEMORY SIZ?` prompt of MITS BASIC, set `strip7out` on the console. Do not set `data_bits = 7` on the board.

When you connect a unit to something other than the console, these settings do not apply to it. Examples are `CONNECT sio0:a socket:2323` and a real serial port. The other end gets the bytes as the guest wrote them, with all eight bits. The serial chapter tells you why, and what to set instead.

`[display]`: the windows that altairsim opens, which are not boards

```
[display]
focus = true
crt   = true
```

`[display]` is like `[console]`. It describes *the window on your screen*, not the board that draws into it. A machine with two video boards still has one operator with one keyboard, so these settings are in one place, not on each board. The settings apply to every window that altairsim opens: the window of each video board, and each terminal window.

The size of a window is different. Each video board opens its own window, so its size is the `width` property of that board. The boards chapter describes `width`.

Key	
`focus`	whether a window comes to the front and gets the keyboard when it opens. Default `false`
`keyboard`	whether the keys that you type in the window of a video board go to the guest: `console` (default) or `none` (display only)
`crt`	show the picture as on the original monitor, with a soft glow and a 4:3 shape, and not as sharp square pixels. Default `false`

`focus`. With `focus = false`, the default, a window opens behind your other windows, and the console keeps the keyboard. You can click in the window and type there. With `focus = true`, a window comes to the front and gets the keyboard when it opens. Use this for a **Sol-20**, where the video window is the keyboard and screen of the machine.

The `focus` setting has an effect only when a window opens. Each time the machine stops, the console gets the keyboard back, so you can type at `altairsim>` at once. This is true on macOS, on Windows and on Linux with X11. On Linux with Wayland, one program cannot give the keyboard to the window of a different program. Click in the console before you type at `altairsim>`.

`keyboard` decides whether a window is a keyboard at all. With `console`, the default, the keys that you type in a window go to the guest. If the machine has a terminal window, the keys go to the unit that the terminal window is connected to. If not, they go to the unit that holds the console. This is correct for a **Sol-20**. Set `none` for a board that only shows a picture, such as a **Dazzler**. The window still shows the picture and still comes to the front, but its keys do **not** go to the guest. They control a joystick, if the machine has a `d7a`. Only `Ctrl-E` works in the window, to stop the machine and give you the monitor. The `keyboard` setting applies only to the windows of video boards. A terminal window always sends its keys to its unit.

`crt` changes how the window shows the picture. It does not change what the machine draws. With `crt = false`, the default, each pixel is a sharp square, scaled by a whole number. With `crt = true`, the window looks like a monitor of the period. The picture is stretched to a 4:3 shape, and the rows are softened into each other. A VDM-1 draws 512×208 and a VDB draws 640×240. Neither picture was square, because the tube stretched it to 4:3. Type `SET DISPLAY crt=on` or `crt=off` at the monitor, and the open window changes at once. With `crt = true`, a window with a `width` is that many pixels wide.

Notes for the operator: `#>`

An ordinary `#` comment is for the person who *reads the file*. A comment that starts with `#>` is for the person who *loads it*. The program prints its text when the machine loads, before the `startup` commands

run. Use it to tell the operator how to use the machine.

```
#> Boots CP/M 2.2 from drive A.
#> Type DIR at the A> prompt, and DIR B: for the blank second disk.

[machine]
name = "cpm22"
base = "default"
startup = ["RUN FF00"]
```

- **Each `#>` line prints as one line.** You can write as many as you want, together or in different places in the file. They print in the order of the file.
- **A `#>` with no text prints a blank line**, so that you can make short paragraphs.
- **A `#>` can follow a setting** on the same line: `name = "cpm22"  #> the buffered variant`.
- **Under `--mcp`, the notes go to stderr.** Stdout carries only the MCP messages there, but you still see the notes in the terminal.
- **A note sets nothing, and `CONFIG SAVE` does not save it.** To keep a note, keep the file that has it, and do not save over that file.

A machine with no base

This small file has every part that a machine needs, and it loads. It has no `base`, so every board is an ADD:

```toml
[machine]
name    = "tiny"

[[board]]
type = "fp"               # the front panel: the sense switches at port FF
id   = "fp0"
sense = 0x00

[[board]]
type     = "8080"         # the processor is a board. The crystal is on it.
id       = "cpu0"
clock_hz = 0              # 0 is as fast as possible. This is the default.

[[board]]
type = "2sio"             # the console board
id   = "sio0"
port = 10                 # HEX: port sixteen

  [board.unit.a]
  baud    = 9600          # DECIMAL: nine thousand six hundred
  connect = "console"

[[board]]
type = "memory"
id   = "mem0"

  [[board.region]]
  type = "ram"
  at   = 0x0000
  size = "16K"

[console]
strip7out = true
```

Saving and loading at the monitor

```
altairsim> SET MACHINE name=mine
altairsim> CONFIG SAVE mine.toml
altairsim> CONFIG LOAD mine.toml
```

`CONFIG SAVE` saves the machine that you are running now. It saves every board and every property, with every change that you made with `SET`. When you load the file, you get the same machine. Give the machine a name with `SET MACHINE name=` before you save it.

`CONFIG SAVE` marks each setting that has the default value. The mark is a comment that names the release the default belongs to:

```
baud = 9600  # default X.Y.Z
stop_bits = 2
```

Here `baud` has the default value, and `stop_bits` does not. A setting without a mark is a value that was set on purpose. The comment does nothing when you load the file. A default can change in a new release, so the mark tells you which release it was true for.

`CONFIG SAVE` saves a new file each time. It does not edit the file that is there. If you save to the name of a machine file that has comments or a `base` line, these things from that file are not in the new file:

- **Each `#` comment and each `#>` note.** The only comments in the saved file are the `# default` marks.
- **The `base` line.** The saved file contains every board in full, so that it does not need a base.

To keep that file as it is, save to a different name.

`CONFIG LOAD` replaces the machine that you have, the same as a machine file on the command line. You cannot undo it, except with a file that you saved. It **loads all of the file or nothing**. The program builds the new machine first. If the file does not load, you keep the machine that you had.

To write a machine file fast, build the machine at the monitor with `BOARDS ADD` and `SET`, and save it. The machines chapter shows the steps, under "From an empty backplane to a machine file".

`STARTUP` builds the `startup` list at the monitor:

```
altairsim> STARTUP ADD MOUNT dsk0:drive0 "CP-M 2.2.dsk"
added: MOUNT dsk0:drive0 "CP-M 2.2.dsk"
altairsim> STARTUP ADD RUN FF00
added: RUN FF00
altairsim> STARTUP
  1  MOUNT dsk0:drive0 "CP-M 2.2.dsk"
  2  RUN FF00
```

`STARTUP ADD` adds a line as you typed it, with its quotes and spaces, because a startup entry is a command line. `STARTUP REMOVE <n>` removes one line, and `STARTUP CLEAR` removes them all. `CONFIG SAVE` saves the list as `startup = [...]`, so the file boots the machine in the way that you tested at the monitor.

Boards

An Altair is a **backplane**, and everything in it is a board. The memory, the serial ports, the disk controller, the front panel and the processor are all boards. Without the boards, only the bus is left.

`altairsim` works in the same way. For this reason, the crystal is a property of the processor *board*, and the sense switches are a property of the *front panel*. You can remove a board, add a different one, and see what the software does.

This chapter starts with the commands that add, remove and show boards. After that, it tells you what each board **is**: what the real hardware was, what it is for, and what can go wrong. **It does not list the properties of each board.** The board reference at the back of this manual lists every property of every board, from the program's own tables.

Add, remove and look at boards

You can do everything by hand that a machine file does to a board.

Command	
`BOARDS`	what is in the backplane
`SHOW BOARDS`	the board types that you can add, one line each
`SHOW BOARD <type>`	the description and settings of one type (add `UNITS` for only the units)
`BOARDS ADD <type> <id>`	add a board
`BOARDS REMOVE <id>`	remove a board
`SHOW <id>`	the settings of one board in the machine, with the values that it accepts
`SET <id> <key>=<value>`	change one setting

```
altairsim> BOARDS ADD virtc vi0
altairsim> SET vi0 rtc_source=line
altairsim> SHOW vi0
```

The keys at the prompt are the keys in a machine file. `SET cpu0 clock_hz=2000000` at the prompt and `clock_hz = 2000000` in a machine file set the same property.

When you have the machine that you want, type:

```
altairsim> CONFIG SAVE mine.toml
```

This writes the machine to a file, and the file loads back as the same machine.

Look at a board type before you add it

`SHOW BOARD <type>` reads the *catalog*. It makes one board of that type, describes it, lists its settings, and removes it again. Nothing is added to the backplane. For a disk controller, the last lines name its units:

```
altairsim> SHOW BOARD dcdd
  dcdd  MITS 88-DCDD: 8" hard-sector floppy, up to 16 drives...
  ...
  This board has units: drive
  SHOW BOARD dcdd UNITS for their properties.
```

Under the legal values of each setting, `SHOW BOARD` gives a `default:` line. It is the value that the setting has when a machine file does not set it. A setting that only shows live state has no default, and has no line:

```
altairsim> SHOW BOARD 2sio
  PROPERTY  HELP
  --------  ----------------------------------------------------------------
  port      Base address. The board decodes four ports: BASE+0 .. BASE+3
            values: 0x0 .. 0xFC
            default: 0x10
```

Add `UNITS` to see only the units. Each unit has a heading with its **kind**, and after the kind, the **command that fills it**:

```
altairsim> SHOW BOARD sol units

  Unit 'serial'  (serial, CONNECT)

  PROPERTY   HELP
  ---------  ----------------------------------------------------------------
  connect    The endpoint on the other end of this line (CONNECT sets this)
  ...

  Unit 'tape1'  (tape, MOUNT)
  mode       Which way the bytes go: play loads from the file, record saves to it
  ...
```

- A **serial** unit is an endpoint, so you `CONNECT` it, to `console`, a socket or a real port.
- A **disk**, **rom** or **tape** unit holds an image, so you `MOUNT` a file in it.
- A **cpu** unit is fixed on the board. It takes neither command, and the heading shows only its kind.

The heading tells you which command a unit takes, before you add the board.

When a *machine file* fills a unit, `UNITS` shows the TOML table and its keys instead. For a disk controller, this is `[[board.drive]]` with its `mount`, `readonly` and `media` keys. You can find the file form as easily as the prompt form.

Contention

When two boards decode the same port, that is **contention**, and it happened on real backplanes too. Both boards answer and both drive the data bus, so the processor reads the wrong value.

The simulator does not stop you from doing this. **It tells you about it.** For example, add an `mds` to a machine that already has a `dcdd`:

```
altairsim> BOARDS ADD mds mds0
mds0: mds added
altairsim> SHOW BUS CONTENTION
  port 08 OUT driven by dsk0 mds0
  port 09 OUT driven by dsk0 mds0
  port 0A OUT driven by dsk0 mds0
```

The output names each port and both boards. You can build a machine that does not work, on purpose, because a wrong setup is part of developing hardware. The program shows you what is wrong.

The boards

The boards are in groups, in the same order as the sections below.

The whole machine

Type	What it is
`fp`	the front panel
`turnkey`	MITS 8800b Turnkey Module: the Altair with no front panel, on one board

Memory

Type	What it is
`memory`	RAM and ROM, as a list of regions
`bankmem`	bank-switched RAM: Vector Graphic, Cromemco 64KZ, North Star HRAM, ExpandoRAM II
`v2z80rom`	the onboard MASTER monitor ROM of the S100Computers V2 Z80 processor board. A paged EEPROM, not a processor

Processors

Type	What it is
`8080`	the MITS 88-CPU
`8085`	an 8085 processor board. It runs all 8080 code, and adds RIM, SIM and the TRAP and RST interrupts
`z80`	a Z80 processor board. The same bus, with a different instruction set

Serial ports and consoles

Type	What it is
`2sio`	MITS 88-2SIO: two serial ports. The usual console
`sio`	MITS 88-SIO: one serial port. The first MITS serial board
`sbc`	SD Systems SBC-100/200: the serial console, timer and parallel port of a Z80 single-board computer
`gsio`	a generic serial board with two channels, set up with straps
`io4`	SSM IO-4: two serial ports and four parallel ports
`propio`	S100Computers Console I/O: a serial console built on a Propeller chip
`pmmi`	PMMI MM-103: a Bell 103 telephone modem on one board

Cassette

Type	What it is
`acr`	MITS 88-ACR: the cassette interface
`uio`	MITS 88-UIO: a serial port and a cassette interface on one board

Printers

Type	What it is
`c700`	MITS 88-C700: the line-printer controller. Capture to a file
`lpc`	MITS 88-LPC: the other line-printer controller, with a line buffer

Parallel and analog I/O

Type	What it is
`pio`	MITS 88-PIO: an 8-bit parallel port, in and out
`4pio`	MITS 88-4PIO: up to four programmable parallel ports
`d7a`	Cromemco D+7A: analog and parallel I/O. Reads joysticks and plays their speakers
`music6`	Newtech Model 6 Music Board: a 6-bit D/A converter and a speaker

Floppy and disk controllers

Type	What it is
`dcdd`	MITS 88-DCDD: the 8″ floppy controller
`mds`	MITS 88-MDS: the 5¼″ minidisk controller
`fdcplus`	FarmTek FDC+: the 1.5 MB floppy, or the serial drive, where the disk images are on a drive server on a serial line
`hdsk`	MITS 88-HDSK: the Datakeeper hard-disk controller
`versafloppy`	SD Systems VersaFloppy I/II: a soft-sector floppy controller. Boots SDOS
`tarbell`	Tarbell #1011: a single-density floppy controller with its own boot PROM. Boots CP/M by itself
`tarbelldd`	Tarbell #2022: the double-density version, for mixed-density disks
`icom`	iCOM FD3712/FD3812: an 8″ floppy controller with its own boot PROM. Boots CP/M and FDOS
`disk1`	CompuPro Disk 1: an 8″ floppy controller with DMA, its own boot EPROM and a bit-banged serial line. Boots CompuPro CP/M 2.2N
`disk1a`	CompuPro Disk 1A: an 8″ floppy controller with DMA and its own boot EPROM. Boots CompuPro CP/M
`mdsa`	North Star MDS-A: a single-density 5¼″ floppy controller in memory, with its own boot PROM. Boots CP/M and North Star DOS
`mdsad`	North Star MDS-A-D: the double-density version, for one-sided and two-sided disks
`16fdc`	Cromemco 16FDC: a floppy controller with a console UART and the RDOS 2.52 boot PROM. Boots CDOS
`64fdc`	Cromemco 64FDC: a floppy controller with a console UART and the RDOS 3.12 boot PROM. Boots CDOS
`dualsd`	S100Computers Dual SD: two microSD cards as CP/M drives. Boots CP/M 3
`dualide`	S100Computers IDE-AB: two CompactFlash cards as CP/M drives. Boots CP/M 3

Video and display

Type	What it is
`vdm1`	Processor Technology VDM-1: memory-mapped video. Needs a display
`dazzler`	Cromemco Dazzler: color graphics. Needs a display
`dazzler2`	Cromemco Dazzler II: the Dazzler with its picture in RAM on the board. Needs a display
`vdb8024`	SD Systems VDB-8024: an 80×24 video terminal on one board. Needs a display
`cadzilla`	CADzilla: an HD63484 ACRTC graphics board with a Bt453 color table. Needs a display
`sol`	Processor Technology Sol-PC: the onboard I/O of the Sol-20, on one board

Interrupts and the clock

Type	What it is
`virtc`	MITS 88-VI/RTC: vectored interrupts and a clock
`ss1`	CompuPro System Support 1: a real-time clock, a serial channel, an interval timer, two interrupt controllers and a math socket
`rtc100`	SciTronics RTC-100: a battery-backed real-time clock, with an optional interrupt each second

Host integration

Type	What it is
`hostbridge`	file transfer to your computer. **Our own board, not a period board**

PROM programmers

Type	What it is
`pb1`	SSM PB1: program a 2708 or 2716 EPROM, and save it as a hex file

The whole machine

The front panel, and the turnkey board that replaces it.

`fp` : the front panel

The front panel is **a board**, because it was one on a real Altair. It plugged into the bus like the other boards. On a real Altair, the panel was how you toggled a bootstrap into memory. Here, the monitor's `DEPOSIT` does that, and the board gives the machine its sense switches.

The SENSE switches, at port `FF`

The sense switches are the eight switches on the left of the address switches, `SA8` to `SA15`. **`IN FFH` reads them.** They are **read-only**. An `OUT FF` does nothing on this board.

`SA15` is the **top** bit of the byte, and `SA8` is the bottom bit. The bits are in the same order, left to right, as the switches on the panel:

```
switch   SA15  SA14  SA13  SA12  SA11  SA10   SA9   SA8
bit         7     6     5     4     3     2     1     0
value    0x80  0x40  0x20  0x10  0x08  0x04  0x02  0x01
```

This makes a period boot procedure easy to follow. "Raise A15 and A11" is `0x80` plus `0x08`, or `%10001000` when you write it to look like the panel.

The switches are important. Period bootstraps read the sense switches to decide **what to boot from**: which device, at which port, and at which speed. For this reason, every tape machine in this package sets them:

```
sense = 0x80        # basic4k: load from the cassette
sense = %10000000   #          the same eight switches, drawn
sense = 0x8E        # ps2:     the 2SIO, and interrupts off
sense = %10001110   #          again, one digit per switch
```

`%` is not special to this board. It works for any number, at the prompt and in a machine file. Do not write `0b` here: the switches are a hex setting, and in a hex number `B` is a digit. *The Monitor* gives the full list of prefixes. Binary is easier to read for eight switches. However you write the value, `SHOW fp0` shows it in hex.

If you set the switches wrong, the loader reads a device that is not there, and it gives no error.

`sense` is a **board property**, because the switches are on the panel. There is no `sense` for the machine, and the program gives an error if you try to set one.

`turnkey`: the MITS 8800bt, on one board

The 8800b "turnkey" system had **no front panel**. One board, the Systems Turnkey Module, did the work of the panel and more. It had the boot PROM, the serial port for the terminal, the sense switches, and a circuit that booted the machine when you turned it on. This board is that board, so a `turnkey` machine has **no `fp` and no separate `2sio`**. All three are on this board. `altairsim turnkey` is the plain machine. Give it a disk and one of the two boot loaders below (`start = "FF00"` for the floppy disk, `FC00` for the hard disk), and CP/M starts when the machine turns on.

It boots itself

There is no front panel to toggle in a bootstrap, so the board has an **Auto-Start** circuit. `RUN 0000` starts the processor at address 0, and the board puts a `JMP` on the bus, so that the boot PROM runs first. This is what the START switch on the panel did. The `start` property is the START ADDR switches: `FF00` runs the floppy loader, and `FC00` runs the hard-disk loader. With `startup = ["RUN 0000"]` in the machine file, this happens when the program starts.

The boot PROM gets out of the way

The PROM is at `FC00`–`FFFF`. **A read there gets the PROM, not the RAM under it,** until the first `IN` from port `FE` or `FF`. The PROM then switches itself off, and the machine has the **full 64 KB** of RAM. For this reason, this machine has 64K, and a front-panel Altair has 56K. The PROM does not use the top of memory all the time. It is a boot device that gets out of the way. The same `IN FF` that reads the sense switches switches it off. Period software, such as Altair BASIC and the CP/M loaders, gets the full 64K in this way without knowing about it.

The console and the sense switches

The serial console is the `tty` unit, at port `10h`. It works like channel A of a `2sio`, so the same software drives it. The sense switches are at port `FF`, as on the front panel. This board uses `FF`, so **do not put an `fp` in a `turnkey` machine**. Both boards would answer the port.

The PROM sockets are a list, like the regions of a memory board:

```
[[board.socket]]
at    = 0xFC00      # socket L1: the hard-disk loader
mount = "builtin:hdbl"
```

Memory

RAM, ROM, and the boards that switch banks of RAM.

`memory`: RAM and ROM

A memory board is **a list of regions**. A real S-100 memory board was the same. It had banks of chips, and each bank decoded the addresses that its jumpers set. A board with 56K of RAM at the bottom and a 256-byte boot PROM at `FF00` was an ordinary board.

For this reason, `default` has one memory board, and that board holds both the 56K and the PROM.

`PHANTOM*`: how a boot PROM gets out of the way

The bus has a line called `PHANTOM*`. **A board pulls this line to switch another board off.** When the processor reads the PROM at `FF00`, the PROM asserts `PHANTOM*`. The RAM board under it stops answering, if its jumper tells it to obey `PHANTOM*`. Two boards decode `FF00`, and only one answers.

This is how an Altair with disks boots. The PROM covers the RAM at the top of memory, and the loader runs from the PROM. After that, **the loader switches the PROM off**, and the RAM under it can be used again. CP/M needs that memory.

Whether a board obeys `PHANTOM*`, and whether it asserts it, are **jumpers** on the board. You set them. If you set them wrong, the machine does not boot, and it does not tell you why. The bus view in the monitor shows you both boards on the same page.

A plain `memory` board has no bank switching. Bank switching is a different board, `bankmem`.

`bankmem`: bank-switched RAM

When 64K was not enough, S-100 makers put several planes of RAM at the same addresses. A write-only **select port** chose which plane the processor saw. Each maker did this in a different way, so `bankmem` is **one board with four decoders**. Its `card` property selects the decoder:

`card`	Real board	Select port	What a write does
`vector`	Vector Graphic 64K	`40`	**one bit for each bank**: `01` →bank 0, `02` →1, `04` →2 … `80` →7
`cromemco64kz`	Cromemco 64KZ / 64KZ-II	`40`	**8-bit mask**: bit *N* turns bank *N* on. **Several banks can be on** (`28` →banks 3 and 5)
`northstar`	North Star HRAM	`C0`	bit 0 = on or off, bits 1–7 = which bank. Banks change **one at a time**
`expandoram2`	SD Systems ExpandoRAM II	`FF`	the byte is a **page number** (see the note below)

`banks` sets the number of planes on the board: up to 8 for `vector` and `cromemco64kz`, 6 for `northstar`, and 10 for `expandoram2`. `fill` and `seed` work as they do on `memory`. The guest writes the select port. At the monitor, you can write it yourself with `OUT`, and `SHOW` lists every plane and which plane is on.

The package has no operating system that uses banks. The quickest way to see the board work is from the monitor:

```
$ altairsim bankmem
altairsim> OUT 40 01
port 40 <- 01
altairsim> DEPOSIT 1000 A0
altairsim> OUT 40 08
port 40 <- 08
altairsim> DEPOSIT 1000 B3
altairsim> OUT 40 01
port 40 <- 01
altairsim> DUMP 1000-1000
1000  A0                                                .
altairsim> OUT 40 08
port 40 <- 08
altairsim> DUMP 1000-1000
1000  B3                                                .
```

`OUT 40 08` selects bank 3, because bit 3 is on. It does not select bank 8. The two `DUMP` commands read different bytes at the same address, because the plane changed.

> **The `expandoram2` page decode is an approximation.** The real board sends the page number through a PROM on the board, and that PROM map is not published in a form that we can copy. For this reason, `bankmem` selects a plain page. The two stock partitions of the PROM are real: `partition = "ex48"` gives 48K of banked RAM and 16K of common RAM at `C000`, and `partition = "ex32"` gives 32K of banked RAM and 32K of common RAM at `8000`. `ram` sets the RAM on the board, and the program calculates the number of banks from it. The other three cards are exact.

`v2z80rom`: the S100Computers V2 Z80 processor board's onboard monitor ROM

This board is not a processor. It is the **monitor ROM** on a real S100Computers V2 Z80 processor board. That ROM is an **8K EEPROM** that holds John Monahan's MASTER V6.6 ROM monitor. A machine that uses this board also needs a `z80` board (below) for the processor. The real board has the Z80 and its own firmware on one board, and here they are two boards.

The EEPROM has **two 4K pages**, and both pages use the addresses `F000`–`FFFF`. A write to port `D3` selects the page. The same write can also switch the EEPROM off, so that the RAM under it can be used. CP/M does this after it boots, to get a flat 64K of RAM. While the EEPROM is on, a read in its window gets the EEPROM, not the RAM.

There is no `BOOT` command. **The monitor boots the machine.** `startup = ["RUN F000"]` starts the monitor. At its `->` prompt, the **`I` command** boots CP/M 3 from a Dual SD card (below). The `dualsd` machine uses this board.

The processor boards

`altairsim` has three S-100 processor boards: `8080`, `8085` and `z80`. This section applies to all three. The sections after it describe each one.

The processor is a board like the other boards. It plugs into the backplane, and you can remove it. With `-n`, you can build a machine with no processor. The processor board decodes no ports and no addresses. It runs the program, and it drives the bus to do that. You can put a `z80` where an `8080` was, and the bus, the other boards and the debugger all work the same.

Every processor board has the same three properties:

- **`clock_hz`**, the crystal
- **`idle`**, which lets the processor rest at a prompt
- **`achieved_hz`**, a read-only value: the speed that the processor reached

The crystal is on the board: `clock_hz`

The crystal is on the processor board, so `clock_hz` is a property of the board, not of the machine. **`clock_hz = 0` is the default, and it means "run as fast as possible".** On a modern computer, that is more than a hundred times as fast as a real Altair.

To run at the speed of a real 2 MHz Altair, type:

```
SET cpu0 clock_hz=2000000
```

To use that speed for one run of a machine file, and leave the file as it is, give the setting on the command line: `altairsim mine.toml --set "cpu0 clock_hz=2000000"`. The machines chapter describes `--set`.

The guest sees the same result at either speed. Each instruction takes the same number of T-states. The crystal changes how the machine feels to you, not what it does. A cassette has its own speed setting, `rate`, which the tapes chapter describes.

There is one exception, at the edge of the machine. A guest counts instructions to measure time. At full speed, a guest's "three-second" timeout passes in a few milliseconds of your time. For this reason, a guest that times something outside the machine, such as XMODEM through a serial port, needs the real crystal. The troubleshooting chapter tells you more.

To see how much time has passed for the guest, type `SHOW CLOCK`:

```
altairsim> SHOW CLOCK
clock  (emulated time -- T-states since POWER, and what they are worth)

  elapsed    0.102953 s   (205905 T-states)
  crystal    2000000 Hz   SET cpu0 clock_hz=N
  pacing     paced -- emulated seconds keep step with real ones

  Elapsed is the GUEST's time, counted from the crystal above: the same
  division a 9600-baud line does to turn a character into T-states. It is
  not how long you have been sitting here, and running flat out is exactly
  when the two differ most.
```

The program counts the T-states since power on, and divides them by the crystal. The result is the guest's time, not your time. A CP/M boot takes the same guest time at full speed and at 2 MHz. Use this number to measure a guest's own timeout.

The `pacing` line tells you if a run keeps real time. It reads `paced` when a crystal is set, and `free` when `clock_hz` is 0. A crystal is always kept, also when you start the program from a script with no terminal. If a script must be fast, leave `clock_hz` at 0, or put `SET cpu0 clock_hz=0` at the start of the script.

`idle`: the processor rests at a prompt

At a prompt, a guest only reads the serial status register again and again, and waits for a key. At full speed, that keeps one core of your computer busy. **`idle` lets the processor rest while the guest reads an empty keyboard.** It is on by default. The busy core then uses a few percent. **The guest cannot see the difference**, because the processor starts again as soon as a byte arrives, before the next read. An XMODEM transfer is correct with `idle` on.

`idle` has an effect only at full speed. When you set a crystal with `clock_hz`, the program already rests to keep the crystal's time, and the guest's time at a prompt stays correct.

`8080`: the MITS 88-CPU

The original processor board. The Altair shipped with it, and the other two processor boards replace it. It has all the properties in the section above, and nothing more.

`8085`: an 8085 processor

The 8085 came after the 8080, and it runs every 8080 program without a change. Of the three processors, it is the nearest to the 88-CPU. Everything in *The processor boards* above applies. This section

gives only what the 8085 adds.

The 8085 adds `RIM` and `SIM`, which read and set the interrupt mask and the SID and SOD serial pins. It also adds its own interrupts, `TRAP`, `RST 5.5`, `RST 6.5` and `RST 7.5`, as well as the 8080's `INTR` line. The documented instructions are correct. This includes the one instruction that is different from the 8080: `ANA` and `ANI` always set the auxiliary carry. The undocumented instructions run too, and set the extra V and K flag bits. `DISASM` marks each undocumented byte, as `DDT` does. The built-in `8085` machine has an `8085`, 64K of RAM and a 2SIO console.

No board drives the 8085's own *pins*: the `SID` and `SOD` serial lines, and the `TRAP`, `RST 5.5`, `RST 6.5` and `RST 7.5` interrupt inputs. The S-100 bus does not carry them. Ordinary interrupts on the `INTR` line work as they do on the 8080.

`z80`: a Z80 processor

The other processor that you can put in the machine. It uses the same backplane, with a different instruction set. Everything in *The processor boards* above applies. The built-in `z80` machine has a `z80`, 64K of RAM and a 2SIO console, so that you can try the processor.

Serial ports and consoles

The boards that have a serial port, for a console, a modem or both.

`2sio`: MITS 88-2SIO

Two **6850 ACIAs**, units `a` and `b`, with four ports at BASE+0 to BASE+3. The base is `10` hex by default, which is where every listing of the period expects it.

This is **the usual console board**, and `default` has one. CP/M and Microsoft BASIC use this board.

The two halves are independent

The two units are two separate chips on the same board. Each unit has its own baud rate, its own connection and its own interrupt setting. For example, unit `a` can be a console at 9600 baud, and unit `b` a modem at 1200 baud. One can use interrupts while the other is polled.

The serial chapter tells you what you can connect a unit to: your terminal, a TCP socket, or a real serial port on your computer, with the modem control lines.

`sio` : MITS 88-SIO

One **COM2502 UART**, unit `tty`, with two ports. This was **the first MITS serial board**. It came before the 2SIO, and the earliest Altair software uses it. The `basic4k` machine uses it.

The port must be **even**. Control is at BASE, and data is at BASE+1.

On the default Rev 1 board, the status bits are inverted

A clear bit means "ready". This is correct for the board, and it is not a bug in the simulator. MITS changed the board at the factory, and on the changed board, "ready" is bit 7 for output and bit 0 for input, both inverted. Every program for the 88-SIO expects this. The I/O routine of `basic4k` tests for zero, and that is correct.

The `rev` property selects the board. `rev = 1` is the changed board, and it is the default. `rev = 0` is the board as it first shipped, which also shows the two signals, not inverted, on bits 5 and 1.

`in_int` and `out_int` set where the receive and transmit interrupts go, as the pads on the real board did.

`sbc` : SD Systems SBC-100/200

SD Systems made S-100 boards for people who wanted a whole computer on as few boards as possible. The **SBC-100** and **SBC-200** are **Z80 single-board computers**, with the processor, memory, a serial console and more on one board. The `sbc` board models the parts that the software uses. They are all in one block of eight ports, `78`–`7F`:

- an **Intel 8251 USART** console, unit `tty`, with data at `7C` and status and command at `7D`
- a **Z80-CTC** at `78`–`7B`, which gives a keyboard interrupt when a byte arrives
- a **parallel port** at `7E`–`7F`. A write to `7F` with bit 1 set switches the onboard memory off
- four sockets for **onboard PROMs**, and **1K of onboard RAM**

The 8251 is not the 6850 that the MITS boards use. Software for a 2SIO cannot use it. The SBC's own **SD monitor** can. The processor is a separate `z80` board, and its `clock_hz` sets the speed. `variant` selects the board: `sbc200` or `sbc100`.

It measures the speed of your terminal

The board has **auto-baud**. Run `altairsim sbc200`, and the SD monitor waits for you to **press Return**. It times the bits of that one character, and sets its own baud rate to match. A terminal at any common speed works. Nothing happens until you press Return. The program is not stopped. It is waiting for that key.

The onboard memory

The jumpers of the real board set where its PROM sockets and its RAM are. The `sbc` board has them as properties:

Property	What it sets	Default
`rom_size`	the size of each PROM: `1K`, `2K`, `4K` or `8K`	`2K`
`bank`	the bank that holds the eight slots. A bank is eight times `rom_size`	`3`, which is `C000`–`FFFF`
`ram`	`true` puts the 1K RAM in the last slot of the bank	`true`
`start`	the auto-start address, a multiple of `1000`	`0000`, which is off

With the defaults, the sockets are at `C000`, `C800`, `D000`, `D800`, `E000`, `E800` and `F000`, and the RAM is at `F800`–`FFFF`. Put a PROM in a socket with a `[[board.socket]]` table. The board refuses an address that is not a socket.

In a machine file:

```
[[board]]
type  = "sbc"
id    = "sbc0"
start = E000

  [[board.socket]]
  at    = E000
  mount = "builtin:msmonr21"
```

The machine needs a `memory` board with RAM under the onboard memory, and that board must have `honors_phantom = "read"`. The onboard memory answers a read. A write also goes to the `memory` board.

Auto-start. A Z80 starts at `0000` after a reset. With `start = E000`, the board makes the Z80 read the PROM at `E000` after a reset, until the PROM reads port `7F`. Then memory is normal again. Thus `RUN` starts the monitor, and `RESET` then `RUN` starts it again.

The `sbc200` machine boots the **SD monitor**. With the **DDBIOS** disk BIOS in a PROM socket and a `versafloppy` controller, the monitor's `C` command boots **SDOS**. See the VersaFloppy section below.

`gsio`: the generic serial board, set up with straps

Most serial boards in this chapter model **one specific chip**, such as the 2SIO's 6850, the SBC's 8251 or the 88-SIO's COM2502. Software must be written for that chip. **`gsio`** is different. It models no specific chip. You **describe** the port with straps:

- which port is status and control, and which port is data
- which status bit means that a byte is waiting (`dav`)
- which status bit means that the transmitter is ready (`tbmt`)
- whether the shared **`inverter_gate`** inverts both bits

Use it for any polled interface that reads a status bit and then a data byte.

The board has **two independent serial channels**, units **`a`** and **`b`**. Each channel has its own straps, `baud` and `connect`, in its own `[board.unit.a]` or `[board.unit.b]` table. By default, `a` uses ports `0` and `1`, and `b` uses ports `2` and `3`.

You seldom set the straps one at a time. A **profile** sets them to match a known board:

- **`sior1`**: MITS SIO Rev 1. This is the default, and the SSM 8080 monitor expects it
- `sior0`: MITS SIO Rev 0
- `tuart`: Cromemco TU-ART
- `imsai-sio2`: IMSAI SIO-2
- `compupro-if2`: CompuPro Interfacer II
- `compupro-ss1`: CompuPro System Support 1

Select a profile for each channel, and then change any strap, as you would move a jumper. The board is polled, with no interrupts. It **only sends and receives bytes**. It does not model word length, parity or stop bits. A board that needs those is a separate board with a full model.

`io4`: SSM IO-4 (2P + 2S)

`gsio` describes a serial port with straps. **`io4`** is a specific board, the **SSM IO-4**. It has **two full-duplex serial channels *and* four latched parallel ports**. Each serial channel is a real UART with **programmable word length (5 to 8 bits), parity, stop bits and baud**. The status byte can be strapped in many ways, as on the real board.

The **two serial channels** are units **`a`** (Serial A) and **`b`** (Serial B). Each has its own `[board.unit.a]` or `[board.unit.b]` table, its own `baud` and its own `connect`. The serial half uses a block of four ports, which must start at a multiple of four (default `0`–`3`). `a` uses ports `0` and `1`, and `b` uses ports `2` and `3`.

Set the straps, or select a profile

The IO-4 had many jumpers, and each jumper is a property here:

- which data bit means that a byte is waiting (`stat_dav`)
- which data bit means that the transmitter is ready (`stat_tbmt`)
- four more status signals: `stat_teoc`, `stat_ror`, `stat_rpe` and `stat_rfe`

- whether the whole status byte is inverted (`invert_status`)
- whether the status port and the data port change places (`port_reversal`)

You seldom set these one at a time. A **`profile`** sets them to match a known host: **`altair-rev1`** (the default, the MITS SIO Rev 1 console that the SSM 8080 monitor expects), `altair-rev0`, `i8251`, `proctech`, `imsai`, and `custom` (all straps free for you to set). Select a profile, and then change any strap, as you would move a jumper.

You can strap the three UART **error flags** (parity, framing and overrun) to the status byte, but they always read as inactive. The serial line carries exact bytes and has no line noise. The current-loop and RS-232 options of the real board are electrical, not programmable, so they are not modeled.

The parallel half

The other half is **four 8212 latched ports**, two for input and two for output, as units **`pa`** and **`pb`** (Parallel A and B). They use their own block of two ports (default `4` and `5`), which must start at an even port. An input port latches a byte on its strobe and sets a service-request flip-flop. A read gets the byte and clears the request. `CONNECT io40:pa …` connects a port to a source or a sink of bytes, like a serial line. The `dav_bit`, `dav_source` and `dav_active_low` straps set up a console with its status byte on one port and its data byte on the other.

If the two blocks of ports overlap, neither half answers on the shared ports. The real board was designed this way. It is not two boards on the same address.

Interrupts, if you strap them

The **W4 header** of the board sets the interrupts, and `io4` follows it. Each serial channel's receive (`rx_int`) and transmit (`tx_int`), and each parallel input (`int`), can go to a vectored interrupt line, to the plain interrupt pin, or to `none`. **There is no software enable.** The strap is the enable, as on the real board. With no straps, the default, the board is polled. A strapped transmit interrupt stays on while the transmitter is idle. A parallel input sets its interrupt on the strobe, also when nothing reads the port. Connect these to a `virtc`, and a received character or a parallel strobe goes to the RST that you strapped.

`SHOW io40` shows every unit, its ports and the live state of its lines.

`propio`: S100Computers Console I/O

A **serial console** from the modern reproduction boards: the S100Computers Console I/O board. It uses a Parallax Propeller chip, not a 6850 or an 8251, with status at `00` and data at `01`. It is a polled console, unit `serial`. You `CONNECT` it to a terminal, a file, a socket or a serial port, like other serial boards. The Dual SD machine uses it as its console.

`propio` is a **preset** of the serial model that `gsio` uses (above). The preset sets the ports of this board, and which status bits mean "receive ready" and "transmit ready". The real board has jumpers, so you can still change each strap. A Console I/O board with different jumpers needs only a property or two, not a new board type.

`pmmi`: PMMI MM-103 modem

A **Bell 103 telephone modem on one S-100 board**. It was the first S-100 modem that was approved for a direct connection to the telephone line. In a real machine, it dialed, answered and carried a serial link at up to 600 baud. It has unit `line`, and four ports from a base that must be a multiple of four. The default base is `C0`. The DIP switch on the real board set it, and the North Star software of PMMI used `E0`.

A read and a write at the same port are different registers:

Port	Write	Read
BASE+0	character format and modem control	UART status
BASE+1	transmit	receive
BASE+2	baud-rate divisor	modem status
BASE+3	the modem chip's control word	nothing that the board drives

The three control registers are **write-only**. The program must keep its own copy of what it wrote, as it did on the real board.

The telephone line

The board does not model the tones of a Bell 103. The line carries exact bytes. You choose what is at the other end of the line:

- **`dial = "host:port"`** is the number that the modem calls. When the guest takes the line off the hook and raises DTR, the board opens a TCP connection to `host:port`.
- **`answer = port`** makes the board answer calls. A TCP connection to that port rings the guest, and the guest answers it as it would answer a telephone call. When the guest drops DTR, the call ends, and the board waits for the next caller.
- On a `dial` or `answer` line, **`telnet`** (on by default) uses the Telnet protocol. A person with a `telnet` client then gets no double echo, and sends one key at a time. Set `telnet = off` for a raw connection, for example to another simulator.
- **`CONNECT`** connects the line to something else, as for other serial boards. With a real serial port, the guest's DTR goes to the port, and the port's CTS, DCD and RI come back in the modem status. `rtsdtr = on` makes RTS follow DTR, for a cable that needs it. With a file or a socket, the modem status always reads "connected and ready".

The board does not decode the dial pulses of the guest into a telephone number. The `dial` property gives the number. The board also raises no interrupts.

`SHOW pmmi0` shows the live character format, baud rate, UART flags and modem lines, with the base address. To try the board, add it to `default`. `BOARDS ADD pmmi pmmi0` adds it at `C0`.

Cassette

Boards that read and write cassette tape.

`acr`: MITS 88-ACR

The **cassette interface**. It is an 88-SIO channel B with an FSK modem, so a byte on the bus becomes a tone on a tape, and a tone becomes a byte again. It has unit `tape` and default port `06`. It runs at 300 baud, because that is what an audio cassette could carry.

With this board, an Altair needs no disk and no PROM. You enter a bootstrap by hand and load the program from a cassette. **The tapes chapter describes this board**: how to load Altair 4K BASIC 3.1, and the `WIND`, `REWIND` and `EXTRACT` commands.

`uio`: MITS 88-UIO

A serial port and a cassette interface on one board. The Universal I/O board is almost a 2SIO channel and an 88-ACR on the same board, with some parts shared. It has the same addresses as the two boards that it replaces: a **6850 serial port** at `10` (unit `serial`) and a **cassette section** at `06` (unit `tape`). Software that expects a 2SIO console and an ACR tape finds both.

It adds **motor control** and a **modulation switch**. `SW-1` selects one of the two encodings of the period: the MITS 300-baud format or the Kansas City standard. The tape section has the same `WIND`, `REWIND` and `EXTRACT` commands and the same tape counter as the 88-ACR.

Printers

Line-printer controllers. Each one sends its output to a file or to a print queue.

`c700`: MITS 88-C700

The **line-printer controller**, an output-only board for an Altair C700 printer. It has unit `prn` and default port `02`, which is the MITS default: control and status at `02`, data at `03`.

The package has no printer, so **`CONNECT` the `prn` line to where you want the output**:

- a file: `CONNECT lpt0:prn out:printout.txt`
- the `console`, to see it print while it runs
- a `socket:`
- a real `serial:` printer
- a **real print queue on your computer**: `CONNECT lpt0:prn printer:linewriter`. This works when your build found a print system. The serial chapter gives the options.

The output is the exact bytes that the program sent, with the control codes. It is not a formatted page.

You can use the board **polled or with interrupts**:

- **Polled:** write a character to the data port (`03`). Before the next character, read the status port (`02`) until bit 0, ACKNOWLEDGE, is set.
- **Interrupts:** set control bit 1 to turn on its **single-level interrupt**. The printer then interrupts the processor each time it takes a byte, and the handler sends the next byte.

The `interrupt` strap selects the line: pin 73 (`int`, the default), a vectored interrupt level (`vi0` to `vi7`), or `none`. `interrupt_after` is switch SW2 #4 of the board. It sets whether the interrupt comes after every character, or only after a CR or LF.

The **`lineprinter`** machine is `default` with one of these boards added. Its output starts on `null`. To capture it, type `CONNECT lpt0:prn out:printout.txt`. To see it, connect it to `console`, which takes the console from the terminal.

`lpc`: MITS 88-LPC

The **other** line-printer controller, for the 88-LP printer. The `c700` drives the C700 printer. The `lpc` has the same two ports: control at an even base and data above it, with the MITS default `02`. It drives the printer as the real one worked, and the output shows the difference.

The C700 **passes every byte through**. The LPC **prints a line at a time**. The guest loads a **6-bit character code** at a time into an **80-character line buffer**. Nothing prints until the guest sends a **PRINT** command, or the buffer is full. **LINE FEED** and **CLEAR** are also commands. For this reason, the output is the printed *page*: the codes changed to characters, one text line for each printed line. On this board, the line breaks are commands, not data.

`CONNECT` its `prn` line to an `out:` file, the `console`, a `socket:` or a real `printer:` queue. The **`lineprinter-lpc`** machine has one at `02`, with its output on `null` until you `CONNECT` it. The board is polled. The interrupt of the real board is not modeled.

Parallel and analog I/O

Parallel ports, one board that also reads analog inputs, and a music board.

`pio`: MITS 88-PIO

An **8-bit parallel port**, in and out. It is the simplest way to move a byte that is not a serial character. It has **two lines that you connect separately**: `out` for an output device (a printer or a socket), and `in` for an input device (a keyboard or another socket). For example, one board can write to an `out:` file and read a keyboard from the `console` at the same time. The default ports are `04` and `05`. The board is **polled**. A byte moves when a driver reads the status port for it.

`4pio`: MITS 88-4PIO

The programmable version of the `pio`. It has up to **four Motorola 6820 PIAs** on one board, and **the guest sets the direction of each port in software**. Each port is a section (`ja`, `jb` and so on), and each section is a line that you can connect. It uses sixteen ports from a default base of `20`. It is polled, like the `pio`.

The `pio` has a fixed eight bits each way. The `4pio` can be set up as the software needs. The `parallel` machine has both boards, and the `pio` sends its output to a file.

`d7a`: Cromemco D+7A

An **analog and parallel I/O board**. It has one parallel port and **seven analog channels**, in a block of eight ports (default base `18`). Each analog channel is an **A/D converter when you read it and a D/A converter when you write it**, in 8-bit two's complement: `00` is 0 V, `7F` is about +2.5 V, and `80` is about −2.5 V. On a real bench, it read sensors and drove instruments.

Here, it is the input and the sound of a **game console**. It reads **one or two JS-1 joysticks**. The X and Y controls go to analog channels. The four buttons of each stick are **active-low** bits in the parallel byte: the low four bits for one stick, and the high four bits for the other. The sticks come from a **USB gamepad** on your computer, or from the **keyboard** (the arrow keys and a few other keys) when there is no gamepad.

`joystick1` and `joystick2` select the device on your computer for each stick. Both are `auto` by default, which gives each stick a **different** gamepad: stick 1 gets gamepad 0, and stick 2 gets gamepad 1. Two controllers work with no setup, and each stick uses the keyboard when its gamepad is not there. `SHOW <id>` shows what each stick uses now (a named controller, the keyboard, or nothing). `SHOW JOYSTICKS` lists the controllers that your computer has.

A machine that uses the sticks can set its video window to be a **display, not a keyboard** (`[display] keyboard = none`). Your keys then move the stick and do not go to a prompt.

The board also plays the **speaker** of each JS-1. A JS-1 has no sound chip. The guest writes a waveform to an analog channel, and the speaker plays the voltage. You hear it on the sound output of your computer.

`speaker1` and `speaker2` give the analog channel of each speaker. The value is `none` or a channel from `1` to `7`, and channel *n* is port base + *n*. The defaults are channel `1` (port `19`) for stick 1 and channel `3` (port `1B`) for stick 2. Some software writes the second speaker to port `1A`. For that software, set the channel to `2`:

```
altairsim> SET d7a0 speaker2=2
```

In a machine file:

```
[[board]]
type     = "d7a"
id       = "d7a0"
speaker2 = 2
```

Sound needs a crystal. At full speed (`clock_hz = 0`, the default), the guest does not keep real time, and the board plays nothing. With a crystal the machine keeps real time, also when you start it from a script with no terminal. Set the speed of the machine to hear the sound:

```
altairsim> SET cpu0 clock_hz=4000000
```

The `dazzler` machine sets 4 MHz already.

`SHOW <id>` has one line for each speaker. The end of the line tells you the state:

The line ends with	Meaning
`-> playing`	The guest changed the speaker output in the last half second of machine time.
`-> silent`	The guest does not drive the speaker now.
`-> silent: the machine has no crystal (clock_hz = 0)`	The machine runs at full speed. Set `clock_hz`.
`-> no sound device`	The program cannot open a sound output on your computer, or this build has no sound.
`-> unwired`	The setting is `none`.

The timing of the board is not complete. A real D+7A holds the processor for a short time on each analog read and write, and the simulated board does not. A guest that makes a tone with a tight loop plays it at a higher pitch than on a real board.

`music6`: Newtech Model 6 Music Board

A **music board**: one output port, a **6-bit D/A converter**, an amplifier and a speaker. The board has no sound chip and no timer. A program makes the sound. It writes a value to the port, waits, writes the next value, and the speaker plays the voltage. The time of the loop is the pitch. You hear it on the sound output of your computer.

The default port is `24`. The board answers at **four addresses**, the port and the next three (`24` to `27`), because the real board does not decode the low two address lines. The `port` setting takes the sixteen addresses that the jumpers of the board can select: `04`, `14`, `24` and so on to `F4`.

```
altairsim> BOARDS ADD music6 music0
altairsim> SET music0 port=44
```

In a machine file:

```
[[board]]
type = "music6"
id   = "music0"
port = 0x44
```

The board uses the **top six bits** of the byte. The low two bits do nothing, so a program that changes only bit 0 or bit 1 is silent. The board has no input port: you cannot read the value back. A reset does not clear the value. Only the power switch does.

Sound needs a crystal. At full speed (`clock_hz = 0`, the default), the guest does not keep real time, and the board plays nothing. With a crystal the machine keeps real time, also when you start it from a script with no terminal. The programs that Newtech supplied with the board are timed for an 8080 at 2 MHz:

```
altairsim> SET cpu0 clock_hz=2000000
```

This short program is the test routine from the manual of the board. It plays a steady tone of about 1009 Hz:

```
altairsim> DEPOSIT 0 97 06 40 05 C2 03 00 2F D3 24 C3 01 00
altairsim> RUN 0
```

`SHOW <id>` gives the value in the latch and a line for the speaker. The end of the speaker line tells you the state:

The line ends with	Meaning
`-> playing`	The guest changed the output in the last half second of machine time.
`-> silent`	The guest does not drive the board now.
`-> silent: the machine has no crystal (clock_hz = 0)`	The machine runs at full speed. Set `clock_hz`.
`-> no sound device`	The program cannot open a sound output on your computer, or this build has no sound.

The amplifier and the volume control of the real board are not modeled. Set the volume on your computer.

Floppy and disk controllers

The disk controllers, from 8-inch floppy disks to CompactFlash. Several of them boot an operating system by themselves.

For most of these controllers, the package has a built-in machine but no disk image. The machine starts with empty drives, and you supply the image. The disks chapter tells you how to mount one.

Disk timing: `timing`

Some of these controllers stop the processor with **wait states** until the disk has the next byte. They are the `tarbell`, `tarbelldd`, `versafloppy`, `16fdc`, `64fdc`, `mdsa` and `mdsad`. The `timing` setting of the board sets how long the wait takes:

- **`full`** (the default): the byte is ready at once, and the wait takes no time. A seek also takes no time. Disk access is as fast as possible.
- **`real`**: the wait takes as long as on the real machine. A seek takes the step time of each track, and each byte comes at the speed of the disk. The processor loses that time, as it did on the real machine.

The bytes on the disk are the same with both settings. Use `real` for a guest that measures time with the disk, or to see how fast the machine really was:

```
altairsim> SET fdc0 timing=real
```

In a machine file:

```
[[board]]
type   = "tarbell"
id     = "fdc0"
timing = "real"
```

With `real`, the processor must be fast enough for the disk, as on the real machine. On the `tarbelldd`, a double-density disk without DMA needs a 4 MHz processor (`clock_hz = 4000000`). At 2 MHz, the guest loses data and reports a disk error.

`dcdd`: MITS 88-DCDD

The **8″ hard-sector floppy controller**. It has up to sixteen drives, and three ports at `08`, `09` and `0A`. **CP/M booted from this board**, and `default` has one.

It can also use the **8 MB medium**, a large format that the same controller can address. You supply the image. The package has no 8 MB disk.

Its status bits are **inverted**, as on the 88-SIO. A clear bit means "ready". The `interrupt` strap sets where the board's interrupt goes.

The machine must run at 2 MHz, or at full speed. The software for 8″ disks (Disk BASIC, Altair DOS, CP/M) reads and writes two bytes each time through its loop, and it times the second byte for a 2 MHz processor. At full speed, the board keeps the same 2 MHz time as the processor, so the disk works. With a faster `clock_hz`, the software reads each byte before it arrives, as on a real Altair with a fast processor, and the disk does not work. The boot PROM then prints `C` again and again.

The disks chapter describes this board: the formats, how to mount a disk, write protection, and the track buffer, which is why you go back to the `A>` prompt before you stop the machine.

`mds`: MITS 88-MDS

The **5¼″ minidisk** controller, with four drives. **It has the same registers as the DCDD**, so a program for one board drives the other. The drives are different:

	`dcdd`	`mds`
Spindle	360 RPM	**300 RPM**
Byte time	32 µs	**64 µs**
Motor	always turning	**stops after 6.4 seconds**

The minidisk motor does not turn all the time. It starts, and if nothing uses the drive, it stops again. The software must handle this. To see it do that, set `motor` to `real`. The default is `free`: the motor is always

at speed, with no wait. The DCDD has no such setting, because its motor never stopped.

You supply the minidisk image. The `minidisk` machine boots its PROM, but the package has no 5¼″ image, so the drives start empty.

It cannot be in the same machine as a `dcdd`

The two boards use the same three ports. This is the MITS address map. A real Altair with both boards would have them both on the data bus at the same time. If you add both here, the bus view names the conflict. Use one or the other.

`fdcplus`: FarmTek FDC+ serial drive and 1.5 MB floppy

The FDC+ is a modern board that replaces the 88-DCDD and the 88-MDS. Its drive type switches select what it does. This board does three drive types: **5**, the 1.5 MB floppy (see The 1.5 MB floppy below), and **6** and **7**, the serial drive.

In its **serial drive** mode it has no disk drive. A **drive server** on another computer keeps the disk images, and the board gets a whole track at a time over a serial line. To the software, the board is an 88-DCDD or an 88-MDS, so CP/M and the boot PROM work as they are.

Use it to use the images on a drive server, to share one set of images between the simulator and a real FDC+ Altair, or to test a drive server.

Property	What it sets
`drivetype`	`7`: the server's disks are 8″ disks, including the 8 MB disk. `6`: they are minidisks. `5`: the 1.5 MB floppy, below. The board reads this at power-on, as the real switches are read.
`connect`	The drive server. `CONNECT fdc0:line` sets it too.
`baud`	The speed of the serial line: 9600, 19200, 38400, 57600, 76800, 230400, 403200 or 460800. Set the same speed on the server. The default is 403200, the fastest reliable speed.
`port`	`08`, or `80`. The board uses four ports.

The server mounts the images, not the simulator. `MOUNT` on `fdc0:line` gives an error. A drive that has no image on the server is not ready, as an empty drive is.

Do not run the machine at full speed. A guest counts instructions to measure time, and CP/M stops looking for a sector after 1.4 seconds of its time at 2 MHz. At full speed, that is a few milliseconds, and a track takes longer than that to arrive. A faster crystal is possible if the line is fast too: 10 MHz with 230400 baud works, and 38400 baud needs 2 MHz.

To use a server on `/dev/cu.usbserial-AL009KFH` at 38400 baud in the `default` machine:

```
altairsim> BOARDS REMOVE dsk0
altairsim> BOARDS ADD fdcplus fdc0
altairsim> SET cpu0 clock_hz=2000000
altairsim> SET fdc0 baud=38400
altairsim> CONNECT fdc0:line serial:/dev/cu.usbserial-AL009KFH
altairsim> RUN FF00
```

The `default` machine's `dcdd` uses the same ports, so remove it first. On Windows, the device is a name such as `serial:COM3`. `CONFIG SAVE` saves the machine to a file, so that you do the setup only once.

A slower line makes a slower disk. At 38400 baud, a track takes about one second. When the machine is not using the disk, the board writes changed tracks back to the server after about one second. It also writes them back before a `DISCONNECT`, a `POWER`, or when you quit.

If nothing comes after `RUN FF00`, the server did not answer. The boot PROM has no time limit: it waits until the drive is ready, and the drive is ready only when the server answers. A real FDC+ does the same. To see what the line does, turn on the board's `error` debug flag:

```
altairsim> SET fdc0 DEBUG=error
```

It reports a server that does not answer (one time, and again when it answers), a track with a bad checksum, a track that did not arrive in full, and each write that the server did not take. The `link` flag shows the same, and also each request and each track. The debugging guide tells you where the reports go.

The 1.5 MB floppy (drive type 5)

With `drivetype = 5`, the FDC+ runs a high-density floppy drive in a format of its own: each track is **one sector of 10,240 bytes**, on both sides of the disk. A disk holds **1,525,760 bytes**, five times an Altair floppy. The board has four drives, `drive0` to `drive3`, and you put a disk image in one with `MOUNT`. The serial line is not used.

A stock Altair boot PROM boots this disk. The PROM cannot read a 1.5 MB disk. But when the PROM asks for a sector, the FDC+ gives it a sector of its own. That sector holds a short loader, and the loader reads the real disk. So `RUN FF00` works with the DBL PROM of the `default` machine.

The machine must run at 2 MHz, or at full speed. The CP/M for this disk moves a track with no handshake: it reads a byte every 17 µs, slightly slower than the disk, and writes one every 14 µs, slightly faster. Those are 2 MHz loops. At full speed, the board keeps the same 2 MHz time as the processor, so the disk works. With a faster `clock_hz`, CP/M reads bytes before the disk has sent them, as on a real Altair with a fast processor, and the disk does not work.

To boot a CP/M disk image for this drive in the `default` machine:

```
altairsim> BOARDS REMOVE dsk0
altairsim> BOARDS ADD fdcplus fdc0
altairsim> SET fdc0 drivetype=5
altairsim> POWER
altairsim> MOUNT fdc0:drive0 CPM22-48K-HDF.dsk
altairsim> RUN FF00
```

The board reads `drivetype` at power-on, so `POWER` comes after it. A disk that is mounted read-only is write-protected. The CP/M for this disk refuses to write to it, and CP/M then shows `Bdos Err On A: Bad Sector`.

`hdsk`: MITS 88-HDSK Datakeeper

A **hard-disk controller**, the "Datakeeper". CP/M boots from it when it does not boot from a floppy disk. It is a separate controller with a **command and handshake protocol** and four **256-byte page buffers**. Unlike the floppy disk boards, **it moves whole sectors for you**. It does not shift bits in real time. It has eight ports, default `A0`–`A7`.

The **HDBL** boot PROM at `FC00` reads the descriptor page of the disk and starts the system. With a CP/M 2.2 disk image mounted, you get an `A0>` prompt. The disk is **read/write**, and CP/M saves to it.

`versafloppy`: SD Systems VersaFloppy I & II

The **soft-sector floppy controller** of SD Systems, built on a **Western Digital FD177x**. It is the board that boots **SDOS**, a CP/M-compatible system, on an SBC-200.

One board type covers both versions. `variant` selects the version:

- **`vfii`** (the default): the double-density VersaFloppy II, with an **FD1791**
- **`vfi`**: the single-density VersaFloppy I, with an **FD1771**

The two versions differ only in the controller chip and a few control bits. They have the same eight ports (default `60`) and the same drivers. Neither version has a boot PROM. The boot BIOS is in a separate PROM, which is the onboard socket on an SBC-100/200.

Add a `vfii`, mount an 8″ double-density SDOS disk, and use the **DDBIOS** of the SBC-200. The disk commands of the monitor then work: **`C`** boots SDOS, and **`R`** and **`W`** read and write sectors.

It checks the density

The density bit of port `63` must agree with the density of the track, as on the hardware. A read at the wrong density ends in **Record Not Found**, and a seek with verify ends in Seek Error. The DDBIOS finds the type of a disk this way: it reads an address at double density first, then at single density. A single-density disk boots, and a double-density disk boots as before.

It cannot format a blank disk

The `Z` command **formats** a disk. On this board, it cannot make a disk from nothing. A disk image holds only the data. It has none of the gaps and address marks that a real format writes between the sectors, so a low-level format has nothing to write. The controller reports a WRITE FAULT. Mount a disk that already has a format, and reads and writes work. For a new SDOS disk, copy a disk that is already formatted. The iCOM board has the same limit. The Cromemco boards (below) can format a blank disk.

`tarbell`: Tarbell #1011 single-density floppy

The **Tarbell Electronics #1011** (July 1977) was the S-100 floppy interface that booted CP/M on many Altair and IMSAI machines. It is a **Western Digital FD1771** soft-sector controller with eight ports, default `F8`. Unlike the VersaFloppy, it has **its own 32-byte boot PROM**, so you do not type a boot command.

Turn the machine on with a disk in drive 0, and the machine boots itself. RESET puts the PROM at address `0000`, over the bottom of memory. The PROM reads the first sector from the disk. When the loaded code runs, the PROM switches off, and CP/M starts. The `tarbell` machine has this board. Mount a CP/M disk in drive 0, and you get the `A>` prompt with no monitor first. With no disk in the drive, the PROM has nothing to load, and it stops. Put a disk in and reset.

The `bootstrap` switch turns the PROM off. The board is then a plain disk controller, for a machine that boots in a different way. It has four drives, which the software selects. The disks are 8″ single density, with 128-byte sectors. The board recognizes a disk by the size of its image.

`tarbelldd`: Tarbell #2022 double-density floppy

The **#2022** (1979–80) is the #1011 with a **Western Digital FD1791**, which reads **double density** as well as single density. It boots in the same way, with the same boot PROM and the same `F8` ports. It boots from a **mixed-density** disk: the first track is single density, so that the boot PROM can read it, and the other tracks are double density. The `tarbelldd` machine boots CP/M 2.2 from a disk of this kind.

This board does everything that the `tarbell` does. It adds double density, and a **DMA register** at port `FD`. With DMA, the board moves data to memory itself, without the processor. Use this board for a double-density Tarbell image, and the `tarbell` for a single-density image. The board recognizes the two kinds by the size of the image.

The board checks the density. The density bit of `OUT FC` must agree with the density of the track, as on the hardware, and a read or a write at the wrong density gives **Record Not Found**. A mixed-density disk needs a boot loader that sets the bit for the double-density tracks. The two double-density CP/M disks of the Tarbell example have one. A disk of your own with an older loader, one that reads track 1 with the bit clear, does not boot: write its boot sector again with a loader that sets the bit.

`icom`: iCOM FD3712 / FD3812 8″ floppy

An **8″ floppy controller** of a different kind. Like the Datakeeper, it uses **commands and a handshake**. It does not shift bits. It holds a whole sector, and the processor moves the bytes through two ports (default `C0` – `C1`). The disk driver of the operating system is in a **boot PROM** in high memory. One board covers both iCOM versions: the single-density **FD3712**, and the double-density **FD3812**. The FD3812 disk is mixed density, with a single-density track 0 and double-density tracks after it. The board recognizes the two by the size of the image.

The `rom` property selects the PROM, and the PROM selects the system that boots:

- `builtin:icom-fd3712-cpm` (the default) boots **CP/M 2.2**, single density, from the PROM at `F000`.
- `builtin:icom-fd3812-cpm` boots **CP/M 2.23**, double density, also at `F000`.
- `builtin:icom-fd3712-fdos` boots iCOM's own **FDOS** from the PROM at `C000`. FDOS is not CP/M. It has a `!` prompt, and its own `LIST`, `EDIT`, `ASMB` and other commands.

The `icom` machine boots CP/M 2.2 as soon as you mount a disk. Reads and writes of a formatted disk work. Like the other controllers here, it cannot format a **blank** disk from nothing.

`disk1a`: CompuPro Disk 1A

The **CompuPro Disk 1A** (1984) is a floppy controller with an **NEC µPD765A** chip, a **DMA** engine and a **boot EPROM**. It has four ports at `C0` – `C3`, and up to four drives. The board moves each byte between the disk and memory itself, without the processor.

The boot EPROM holds sixteen 512-byte routines. The `routine` switch picks one, and RESET puts it at address `0000`. Routine 0, the default, reads track 0 of drive 0 into memory at `0100`. It then switches the EPROM off and starts the loader that it read. The `boot` switch turns the EPROM off, for a machine that boots in a different way.

The `disk1a` machine has this board, a Z80 processor, 64K of RAM and a **System Support 1**. The console is the serial port of the System Support 1. The machine starts the EPROM at once. With no disk in drive 0, routine 0 tries again and again. To boot a CompuPro CP/M disk:

1. Press `Ctrl-E` to get the monitor.
2. Mount the disk in drive 0.
3. Run from `0000`.

```
altairsim> MOUNT fd0:drive0 "CCPM22.DSK"
altairsim> RUN 0
```

The board recognizes a disk by the size of its image. An 8″ CompuPro disk has a single-density track 0, which the EPROM reads, and its other tracks can be double density. A disk can have one side or two. A smaller image that has no known size is a blank disk. CompuPro's `FORMAT.COM` formats it.

A blank disk needs `media`, which gives the diskette its size and sides. `8ds` is an 8″ disk with two sides, and `5ss` is a 5.25″ disk with one side. Without `media`, the board refuses a blank, and the error names the values to set. This command puts a blank 8″ two-sided disk in drive 1:

```
altairsim> MOUNT fd0:drive1 BLANK.DSK CREATE media=8ds
```

In a machine file:

```
[[board]]
id = "fd0"

  [[board.drive]]
  unit   = 1
  mount  = "BLANK.DSK"
  media  = "8ds"
  create = true
```

`create` makes the empty file `BLANK.DSK` if it is not there. The drive keeps its `media` when `MOUNT` puts a different disk in it.

Three 5.25″ formats have 327,680 bytes, and two have 655,360 bytes. The board refuses an image of these sizes, because the size does not tell it the format. Set `media` to the whole format, such as `5ds-dd512` for two sides of 8 × 512-byte sectors. The error message names the values that fit the image.

These things are not part of the board:

- **A 5.25″ disk boots from unit 2.** Routines 2 and 3 of the EPROM read it.
- **Only routines 0–3 run.** The other routines are for the 8086, 68000 and 32016 processors.
- **CompuPro CP/M 2.2Q does not run.** Its BIOS needs the 8088 of the CompuPro CPU 8085/88 board.

`disk1`: CompuPro Disk 1

The **CompuPro Disk 1** (1981) is the board that came before the Disk 1A. It has the same **NEC µPD765A** chip, **DMA** engine and four ports at `C0`–`C3`, and up to four drives. Port 3 is different. On the Disk 1, bit 7

of port 3 is a serial line with no UART. A guest program sends and receives each bit itself, with a delay loop for the timing.

The boot EPROM holds eight 256-byte routines. The `routine` switch picks one, and RESET puts it at address `0000`. The built-in EPROM is a copy of a factory EPROM. Routines 4–7 are for the 8080, 8085 and Z80, and routine 4 is the default. Each reads the start of track 0 of an 8″ disk in drive 0 into memory at `0100`, switches the EPROM off, and starts the loader. Each routine passes a different console code to the loaded system:

Routine	Console of CompuPro CP/M 2.2N
4	The Interfacer at `00`/`01`
5	An Interfacer at `10`/`11`
6	The serial channel of the System Support 1
7	An Interfacer 3 or 4 at `10`

Routines 0–3 are for the 8086 and do not run here.

The `disk1` machine has this board, a Z80 processor, 64K of RAM, a **System Support 1** and an Interfacer I console. The console is the first serial port of the Interfacer I, at ports `00` and `01`, as the CompuPro CP/M 2.2N BIOS expects. To boot a CompuPro CP/M 2.2N disk:

1. Press `Ctrl-E` to get the monitor.
2. Mount the disk in drive 0.
3. Run from `0000`.

```
altairsim> MOUNT fd0:drive0 "CPM22N.DSK"
altairsim> RUN 0
```

To use the System Support 1 as the console, set routine 6 and move the console:

```
altairsim> SET fd0 routine=6
altairsim> CONNECT if1:a null
altairsim> CONNECT ss1:serial console
```

Each boot logs one line, `fd0: serial: a break on the line (no stop bit); nothing sent`. The CP/M 2.2N BIOS sets bit 7 of port 3 to 1, and 1 holds the serial line at SPACING. Nothing is connected to the line, so nothing is lost. To use the line, connect its `serial` unit:

```
altairsim> CONNECT fd0:serial telnet:2323
altairsim> SET fd0:serial baud=1200
```

Set `baud` to the rate of the guest's delay loop.

To format a blank disk, give its drive `media = "8ds"`, an 8″ disk with two sides. Without `media`, the board refuses a blank, and the error names `8ss` and `8ds`. This command puts a blank 8″ two-sided disk in drive 1:

```
altairsim> MOUNT fd0:drive1 BLANK.DSK CREATE media=8ds
```

In a machine file:

```
[[board]]
id = "fd0"

  [[board.drive]]
  unit   = 1
  mount  = "BLANK.DSK"
  media  = "8ds"
  create = true
```

`create` makes the empty file `BLANK.DSK` if it is not there. `FORMAT.COM` formats drive B with two sides, and `STAT B:` reports a 1200K drive.

These things are not part of the board:

- **The board runs 8″ drives only.** A Disk 1 ran 5.25″ drives only after a change to the board's parts and wiring. The board refuses a 5.25″ disk image.
- **The M-Drive/H RAM disk is not part of the machine.** The 2.2N BIOS reports `0 M-Drive/H` and continues.

`mdsa` and `mdsad`: North Star MDS-A and MDS-A-D

The **North Star Micro-Disk System** was a 5¼″ hard-sector floppy system for 8080 and Z80 machines. The `mdsa` is the single-density controller of 1977. The `mdsad` is the double-density controller of 1978, which also reads and writes single density.

These boards have no ports. Each one decodes a 1 K block of memory, from `E800` to `EBFF`. A program gives a command when it reads an address in the block. The low byte of the address is the command, or the data byte to write. A read of `EB90` on the `mdsa` starts the drive motors and returns a status byte.

A `DUMP` of the block gives commands to the board. `DUMP` and `EXAMINE` read memory as the processor does, and on this board a read is a command. `DISASM` and the debugger look at memory without a read. They show the boot PROM, and `FF` for the other addresses.

Each board has its own **256-byte boot PROM** in the block. The `northstar` machine has an `mdsa`, and the `northstardd` machine has an `mdsad`. The drives start empty, and you supply the image. To boot a disk:

1. Start the `northstar` machine.
2. Mount the image in the first drive: `MOUNT fd0:drive0 <file>`.
3. Start the boot PROM: `RUN E900`.

	`mdsa`	`mdsad`
Start the boot PROM	`RUN E900`	`RUN E800`
Drives	3	4
Density	single	single and double
Sides	1	1 or 2
The motors stop after (`motor = real`)	3.2 seconds	9.6 seconds

`drive0` is the drive that North Star software calls drive 1. The units of a board start at 0, as on every controller. North Star DOS and the boot PROM start at 1.

The board recognizes a disk by the size of its image. An image holds only the data of each sector:

Format	Sectors	Bytes/sector	Sides	File size	Board
`sd`	35 × 10	256	1	**89,600**	`mdsa`, `mdsad`
`dd`	35 × 10	512	1	**179,200**	`mdsad`
`quad`	35 × 10	512	2	**358,400**	`mdsad`

The boot PROM of the `mdsad` boots a double-density disk. To boot a single-density disk, use the `mdsa`. The `mdsad` reads and writes a single-density disk in a drive that it does not boot from.

An empty file is a blank disk. On the `mdsa`, a blank disk is single density. On the `mdsad`, a blank disk gets its density from the first sector that the guest writes, which is what the format command of the guest decides. On the `mdsad`, a file of any other size does not mount. Set `media` on the drive to give the format.

One image has one density. The `mdsad` does not write a single-density sector to a double-density image, and the monitor tells you when the guest tries. A write-protected disk shows in the status byte of the board, so the guest knows before it writes.

`motor` works as it does on the `mds`. With `free`, the default, the motors run until the guest stops them. With `real`, they stop after the time in the table, and the guest starts them again. `interrupt` sets where the interrupt of the board goes. The board makes an interrupt at each sector, 50 times a second, only when the guest arms it. North Star DOS and CP/M do not arm it. `base` moves the block, but the boot PROM in the board is the standard part, which runs only at `E800`.

Double-density North Star DOS and the memory at `2000`

Double-density North Star DOS keeps the track of drive 1 in the byte at `2000`. It reads that byte before it writes it, and the boot PROM does not write it. If the byte is `80` or more, the DOS moves the head to a wrong track, and the boot fails. A second `RUN E800` then boots, because the DOS wrote the byte during the first boot.

The `northstardd` machine starts with all its memory at zero, so the first boot works. If you add an `mdsad` to a different machine, set `fill = "zero"` on the memory board, or start the boot two times. CP/M and single-density North Star DOS do not have this problem.

`16fdc` and `64fdc`: Cromemco 16FDC and 64FDC

Cromemco's double-density floppy controllers. Each board has three things on it:

- a **Western Digital FD1793** floppy controller, for single and double density, with up to four drives. Its registers are at `30`–`34`.
- a **TMS 5501** console UART, unit `tty`, at `00`–`09`
- an **RDOS** boot PROM at `C000`. The 16FDC has 4K of RDOS 2.52. The 64FDC, from 1983, has 8K of RDOS 3.12, at `C000`–`DFFF`.

`OUT 40H` switches the PROM off, and RESET switches it on again. With the `bootstrap` strap on (the default), RDOS boots the disk. With it off, RDOS gives its monitor prompt.

Both boards boot Cromemco **CDOS**. The package has no CDOS disk and no built-in machine for these boards, so you add the board and supply the disk. **These boards can format a blank disk.** Mount a blank image, and the guest's own format program writes the tracks.

RDOS boots drive A. To boot a different drive, set `boot_drive` to `B`, `C` or `D`. This sets the boot-drive switches of the board. RDOS reads them for the automatic boot, and for its `B` command when you give no drive letter:

```
altairsim> SET fdc0 boot_drive=B
```

In a machine file:

```
[[board]]
type       = "64fdc"
id         = "fdc0"
boot_drive = "B"
```

The boards check the density. Bit D6 of port `34` must agree with the density of the track, as on the hardware, and a read or a write at the wrong density gives **Record Not Found**. A 5.25-inch double-

density track that a program formats is recorded as double density. Both boards use the same density bit at the same 250 kbit/s for 5.25-inch single and double density, so the bit is the only thing that tells the two apart.

Port `04` bit D6 reads `0` on the 16FDC. On the 64FDC it always reads `1`. CDOS reads this bit for each 8-inch drive: a `0` marks a PerSci drive, which CDOS homes with the `¬RESTORE` line of port `04` (D3). The 64FDC has no such line, so CDOS homes its drive with the disk chip.

The boards are polled. They deliver no interrupts. The other switches of the boards are fixed: the console has a preset baud rate, and RDOS boots from a floppy disk.

`dualsd`: S100Computers Dual SD

A **modern** disk controller among the period boards. The S100Computers Dual SD board puts **two microSD cards** on the S-100 bus as drives with 512-byte sectors, and a Z80 machine runs **CP/M 3** from them. Like the iCOM and the Datakeeper, it uses **commands and a handshake**. It has two ports (default `80`–`81`), and a microcontroller on the board does the card I/O. Each SD card is one CP/M drive. The **socket** of the card gives the drive letter: socket 1 is A:, and socket 2 is B:.

It has no boot PROM. The monitor on the processor board boots it. For this reason, the `dualsd` machine has three boards: a `z80` for the processor, the `v2z80rom` monitor EEPROM, whose `I` command reads CP/M 3 from the card, and this controller. Start the machine, type `I` at the `->` monitor prompt, and CP/M 3 starts at `A>`.

Mount a card in both sockets. The CP/M 3 boot ROM checks that each socket has a card before it starts. With a card in socket 1 only, the loader boots and then stops. Put the bootable system card in socket 1, and a blank card in socket 2.

A card is a raw **`.img` file, with a `.geo` file beside it** that gives the true size of the card. The `.img` can be **shorter** than the card, with only the live file system in it. Every sector after its end reads as an erased card does, as all `FF`. For this reason, a card that is hundreds of megabytes on real flash can be a few megabytes here. `MOUNT … CREATE` makes a **blank** data card, an empty `.img` and its `.geo`, for the guest to format. A *bootable* card must come from a real image, because nothing can make its system tracks. You supply the images.

`dualide`: S100Computers IDE-AB (CompactFlash)

The other half of the same S100Computers board. The Dual SD part drives microSD cards. The **IDE-AB** part drives **two CompactFlash cards**, through an 8255 parallel port that connects to the IDE bus of a CF card (default ports `30`–`34`). Its two cards are CP/M drives A: and B:, and it boots the same **CP/M 3**. It stores data on a card in the same layout as the SD part, so **a card image works on both boards**.

It has no boot PROM either. The `dualide` machine has the `z80`, the `v2z80rom` monitor EEPROM and this controller. At the `->` monitor prompt, the **`P` command** (not the `I` of the Dual SD) reads CP/M 3 from CompactFlash drive 0, and CP/M starts at `A>`.

The cards are the same as on the Dual SD board: a raw **`.img` with a `.geo` file beside it**. The image can stop after the live file system, with `FF` after its end. `MOUNT … CREATE` makes a blank data card, and a bootable card must come from a real image. The **`dualidesd`** machine has the **whole board** at once: CompactFlash as A: and B:, and microSD as C: and D:, with one CP/M 3 system on all four drives.

Video and display

Boards that show a picture in a window.

Each video board opens its own window. `[display]` in the configuring chapter sets which window gets the keyboard, and how the picture looks.

Closing a video window stops the machine. It does not quit the program. The close box does what `STOP` does. The guest stops at an instruction boundary, and you get the monitor prompt, with the machine where it was. `RUN` continues in the same window. `QUIT` leaves the program.

The window size: the `width` property

Every video board (the VDM-1, the Dazzler and the VDB-8024) has a **`width`** property. It sets the width of the window in pixels. `auto`, the default, makes the window about **half as wide as the screen**. A number such as `width = 1024` asks for that many pixels. **The height follows from the shape of the board's picture.** With the default look, the program draws each pixel of the board as a whole number of screen pixels, so that the picture stays sharp. A thin dark border fills the rest. With the period look (`[display] crt = true`), the picture fills the width that you asked for, because a soft picture has no square pixels to keep sharp. A width that is too big for the screen is made smaller.

The built-in `terminal` window (see the serial chapter) sets its size in the same way, with a `width=` option in its connect string, not a board property.

`width` is a property of the board, not of `[display]`. On real hardware, each board had its own video output and could drive its own monitor. The same is true here. Add a VDM-1 and a Dazzler to one machine, and you get two pictures in two windows. The keyboard is still shared. Whichever window you type in, the keys go to the machine.

A stopped machine's window does not redraw

The window stays live when the machine is stopped. You can move it, and you can close it. If you close it at the monitor prompt, `RUN` opens a new one the next time a program draws. A stopped machine does **not** redraw the picture, because the running machine draws it. This has two results:

- **A change such as `SET vdm0 video=reverse` does not show** until you type `RUN` again. The program keeps the setting, but nothing redraws the screen until the machine runs.
- **The cursor does not blink** while the machine is stopped. To see it blink, use `sol20`, where SOLOS runs in a loop and does not halt.

The `vdm1` demo halts after it draws its banner, so its window belongs to a stopped machine.

`vdm1`: Processor Technology VDM-1

Memory-mapped video. A 1K screen of **16 rows by 64 columns** is in the machine's own address space, at `CC00` by default. A program puts a character on the screen when it *stores a byte*. It needs no port and no driver. For this reason, the board is fast, and it needs no `CONNECT`, because the screen is memory.

One port (default `CC`) does the rest. A write to it sets which row is at the top of the screen. This is how the VDM-1 scrolls: the text does not move, but the window on it does. A read of the port gives two timing bits. The software uses them to avoid writing while the beam draws.

Bit 7 of each byte is the **cursor and blink** flag, not part of the character. For this reason, the board draws 128 characters from a real character ROM, not 256.

Two machines use it:

- `vdm1`: an Altair with a VDM-1, and a demo that draws on it
- `cuter`: the period CUTER monitor, with its own VDM-1 driver

`dazzler`: Cromemco Dazzler

The **first color graphics board for the S-100 bus**. The VDM-1 shows text, and the Dazzler shows a **picture**. It does this in the same way, from a **frame buffer in the machine's own RAM**. You point the board at a block of 512 bytes or 2 KB, at any 512-byte boundary, and it shows that memory on the screen. A program draws when it *stores bytes*, with no port in the inner loop.

Two ports (default `0E` and `0F`) set the rest. The first turns the board on or off, and sets where the frame buffer is. The second sets the format: the resolution, the size and the color. This gives four modes: **32×32 or 64×64** elements in color or grey, and **64×64 or 128×128** on/off elements, in **16 colors** or 16 greys.

The `dazzler` machine is the plain board, for you to build on. A 64×64 picture is very small, so the board's `width` property (above) makes the window about the size of a VDM-1 window.

`dazzler2`: Cromemco Dazzler II

The Dazzler II is **a modern board**, not a period product. It works like the Dazzler: the same two ports (`0E` and `0F`), the same four modes and the same picture. The difference is where the picture is kept. The Dazzler reads its picture out of the machine's RAM. The Dazzler II keeps the picture in **4 KB of RAM on the board**. It watches the processor's writes to memory, and it keeps a copy of each one that is in its 4 KB window. The window starts at the address you set with `OUT 0E`.

Use `dazzler2` to **test a program for a real Dazzler II**. The board shows what the real board shows, and the Dazzler hides these things:

- The board RAM has no defined contents until the program sets the base. A program that draws its picture *before* `OUT 0E` shows random data. The `seed` property sets the data, so the same seed gives the same picture each time.
- The picture can be at an address that has no RAM in the machine, such as a ROM or an empty slot. The Dazzler shows nothing there.
- The board shows 2 KB of its 4 KB. The `page` property is `0` for the first half, or `800` for the second half. On the board, this is jumper P18 pins 31 and 32.
- A read of the status port (`IN 0E`) gives `0` in bits 0 to 5. The Dazzler gives `1`. A program that waits for the value `3F` can wait for ever.

The board does not model the second color map, the page bit set by the program, or the joystick circuit that is also on the real board. The `d7a` board serves the joystick. For the Cromemco games, use `dazzler`.

`cadzilla`: an HD63484 ACRTC graphics board

This board is **a new custom product based on period components**, not a period product. It uses two real chips from the middle of the 1980s. The **Hitachi HD63484 ACRTC** is a CRT controller with a *drawing processor* and up to 2 MB of its own frame memory. The **Brooktree Bt453** is a RAMDAC: a color table of 256 entries, with three video DACs on one chip.

The VDM-1 and the Dazzler show a picture from the machine's RAM. The CADzilla board keeps its picture in its own memory, and the processor never addresses a pixel. The guest **draws with commands**, for example "move here", "draw a line to there" and "clear this block". The guest writes each command into the FIFO of the ACRTC, one word at a time, and the chip draws.

The board decodes one block of 8 I/O ports from `port` (default `70`):

Port	What it is
`port`	The ACRTC address register (write) and status register (read).
`port+1`	The board's own MODE register, write only. It holds the sync polarity, the access mode of the board's fetch logic, and the two overlay bits `OLEN` (bit 3) and `OLSEL` (bit 4). A bus reset sets it to 0.
`port+2`	The ACRTC control register that the address names, one byte at a time. The high byte is at an even address, and the low byte is at the odd address.
`port+3`	Not decoded. On the real board, it is the high byte of a 16-bit transfer. An 8080 or Z80 does only 8-bit transfers.
`port+4` to `port+7`	The Bt453: the address register, the color table (red, green, then blue), the address again, and the overlay colors.

The timing and display registers of the ACRTC increment the address after each byte. For this reason, one address write loads a whole block of these registers.

The ACRTC sends out an 8-bit value for each pixel. The Bt453 changes that value into a color from its table. A guest sets its colors when it loads the table. A new table changes the colors of the picture with no redraw.

The monitor is part of the board. The `mode` property sets a fixed-frequency VESA display: `640x480`, `800x600` or `1024x768` (the default). Set it in the machine file or with `SET`. The window always has the size of the mode, as the frame of a real monitor does. The timing registers of the ACRTC set where the picture is in that frame. A picture that starts where the back porch of the mode ends fills the frame. A picture that starts one cycle early or late moves, and its edge is cut off.

The shift register of the board is wired for **8 bits per pixel and 8 words for each fetch**. A guest must set the ACRTC to 8 bits per pixel and an address increment of +8. Each fetch reads 8 words that start at a multiple of 8, so the board ignores the low 3 bits of a start address. `SHOW <id>` has a `wiring` line. The line shows `ok`, or it names the setting that is wrong.

The overlay is a second picture of 1 bit for each pixel, in its own memory. Where an overlay bit is 1, the Bt453 shows its **overlay color 2** in place of the color from the table. The overlay shows only when the `OLEN` bit of the MODE register is 1. To draw the overlay, the guest uses an ACRTC screen with the CHR bit set in its memory width register. The board sends each drawing access to that screen to the overlay memory, not to the frame memory. The `OLSEL` bit of the MODE register selects which half of the overlay memory the guest draws. Do not enable that screen for display: the board then shows frame memory in its area, and the `wiring` line names the screen. To set overlay color 2, write `02` to `port+4`, then red, green and blue to `port+7`.

The board needs a display. An SDL3 build opens a window with the board's id as its title. A build with no display runs in the same way and shows nothing. The frame memory is 2 MB, and you cannot change it.

`SHOW <id>` gives these straps and live values:

Name	What it is
`port`, `mode`, `draw_rate`, `width`	Straps. You set them in the machine file or with `SET`.
`interrupt`	A strap. It sets the S-100 line for the IRQ* output of the board: `none` (the default), `int`, or `vi0` to `vi7`. On the real board, it is switch 8 of SW1.
`video`	Live. `on` when the ACRTC shows a picture.
`picture`	Live. The size of the picture, and its position in the frame.
`wiring`	Live. `ok`, or the ACRTC setting that does not agree with the board.
`hspol`, `vspol`, `amode`, `olen`, `olsel`	Live. The bits of the MODE register.
`status`	Live. The ACRTC status register.
`irq`	Live. Whether the board asserts an interrupt now.

The `draw_rate` strap sets the drawing speed. With `full` (the default), the ACRTC completes each command immediately. The write FIFO is always empty, and the guest never waits for the chip. This is the fastest setting when you develop a program. With `real`, each command takes the time that the Hitachi data sheet gives for it. The ACRTC draws only in the memory cycles that the display does not use. These depend on the access mode and on the priority bit of the operation mode register. While the chip draws, the next command words stay in the write FIFO. The FIFO status bits and the command-end bit (CED) change at the times that the real chip changes them. Use `real` for a guest that is sensitive to time, for example a game:

```
SET cad0 draw_rate=real
```

The ACRTC clock comes from the pixel clock of the `mode`. It is the pixel clock divided by 8 in single access mode, or by 4 in interleaved access mode. The processor's `clock_hz` does not change the drawing time that the guest sees. At each `clock_hz`, a drawing command takes the same number of processor instructions. With `clock_hz = 0`, the host does not wait, and the guest sees no change.

The `cadzilla` machine is the board alone, with a console for you to type at. The board does every drawing command of the ACRTC: lines, rectangles, circles, ellipses, arcs, area paint, patterns, and block and area copies.

`vdb8024` : SD Systems VDB-8024

An **80-column by 24-line video terminal on one board**. The `sbc` board gives an SBC-100/200 a serial port for a terminal. The VDB-8024 gives it *the terminal itself*: a complete display with a keyboard, on the backplane.

It is not memory-mapped. None of its screen is in the machine's address space. To the computer, it is **two I/O ports**, a status port and a data port at `00` and `01`, that work like a terminal on a line. The program reads the status to see whether a key is waiting or the display is ready. It writes a character or a control code to the data port, and it reads a typed key from the same port. The screen, its memory and its own processor are all behind those two ports, as on the real board. The real board had its ports fixed at `00`. The board here has a `port` property so that you can move them, as the `sol` does below.

The board is polled by default. The `interrupt` strap (`vi0` to `vi7`) sends a keyboard interrupt to a vectored interrupt line, for the SBC-200's CTC. The SD video CBIOS needs this.

It runs `sdmonv21`, the video version of the SD monitor. It is the same monitor as in the serial `sbc200` machine, with the same commands and the same `.` prompt. You **do not press Return first**, because the VDB has no serial speed to measure. The prompt is on the screen as soon as the machine starts:

```
$ altairsim sbc200v
```

This gives the monitor's `.` prompt **in the video window**.

Like a Sol-20, **the window is the console.** The machine sets `focus=on`, so the window keeps the keyboard while the guest runs. Keys typed in the window and at the terminal go to the monitor together. The characters come from the board's own character font, with true descenders on `g`, `j`, `p`, `q` and `y`, as the font PROM of the hardware drew them.

The board obeys the same control codes as its firmware: carriage return, line feed, backspace, tab, cursor up and right, clear screen and home, and the `ESC` sequences that move the cursor and erase to the end of a line or of the screen. A full line wraps, and a line feed on the bottom line scrolls the page up.

`sol` : Processor Technology Sol-PC

The **onboard I/O of the Sol-20, as one board**, because it was one board on a real Sol-20. The Sol was not an Altair with boards in it. It was one machine, with the serial port, keyboard, parallel port and cassette interface all on the processor board, at `F8`–`FE`. On the real hardware, those addresses were fixed. Here, the board has a `base` property so that you can move it. This is the one change from the hardware, and the reference records it.

You reach the parts of the board as units. `serial`, `printer` and `keyboard` are lines that you `CONNECT`. `tape1` and `tape2` are cassette decks that you `MOUNT`. The keyboard is connected to the console by default. It takes what you type, from the video window when there is one, or from your terminal.

The special keys of the keyboard

The Sol keyboard has eight keys that send codes with no ASCII equal. You use them to control SOLOS and the screen:

Key	Sends	What it does	Press	Or type
`←`	`81`	Cursor left one	←	Ctrl-A
`→`	`93`	Cursor right one	→	Ctrl-S
`↑`	`97`	Cursor up one	↑	Ctrl-W
`↓`	`9A`	Cursor down one	↓	Ctrl-Z
`HOME CURSOR`	`8E`	Cursor to the top left, screen untouched	Home	Ctrl-N
`MODE SELECT`	`80`	Return to the command mode, restarting the command line	F1	Ctrl-@
`CLEAR`	`8B`	Erase the screen, cursor home	F2	Ctrl-K
`LOAD`	`8C`	Nothing. Neither SOLOS nor CONSOL uses it	F3	none

The `Press` column works only in the video window. When the window has focus, your arrow keys, Home, and F1 to F3 send these codes. They cannot work from a terminal. There, an arrow key or a function key sends an escape sequence, not one byte, and the guest needs `ESC` as a character, so the program cannot safely read those sequences. From a terminal, use the last column.

A PC keyboard has no `MODE SELECT`, `CLEAR` or `LOAD` key, so **F1** is `MODE SELECT`, **F2** is `CLEAR`, and **F3** is `LOAD`. On a Mac, the function keys reach the window only if the system sends F1, F2 and so on as plain function keys. If not, hold `fn`.

The last column is how the hardware worked. The code of each special key is `80` plus the control code for the same action, because the display driver of SOLOS clears the top bit of each character before it uses it. For example, `CLEAR` and Ctrl-K go to the same routine. The command reader clears the top bit too. For this reason, a NUL byte is `MODE SELECT`. Type Ctrl-@ (Ctrl-Space on many keyboards), and SOLOS drops the line that you were typing and gives a new prompt.

The console is 8-bit clean. If you can send the byte itself, for example with a paste, a script or a terminal macro, the real code works too.

One register (`FA`) gives the state of *all* of these parts at once, with **mixed polarity**. The keyboard and parallel bits are active-low, and the tape bits are active-high. This is what the hardware did, and the period software inverts the bits that it needs.

The cassette decks

The Sol's cassette interface is on the main board, and it has **two decks**, `tape1` and `tape2`. The 88-ACR has one. Everything in the tapes chapter applies to them, with a different unit name. Always give the name of the deck. The program refuses a plain `WIND sol0`, because it does not know which tape to move.

```
altairsim> MOUNT sol0:tape1 "mytape.tap"
altairsim> SET sol0:tape1 mode=record
altairsim> REW sol0:tape1
```

There are three differences from the ACR, and each one comes from the hardware:

- **The Sol controls the motors.** `OUT 0FAh` starts and stops each deck, and SOLOS does this for you. `SAVE` starts the deck, writes, and stops it. For this reason, a Sol tape plays only while the guest runs it, and a deck with its motor off gives nothing. When the motor stops, the board writes the recording out. The 88-ACR cannot see the deck, so it cannot do this. The Sol still cannot **wind** the tape by itself, because a motor line says only "turn", not which way. You use `WIND` and `REWIND` for that.
- **The guest sets the speed.** The ACR's 300 baud is fixed. The Sol's cassette runs at 300 or 1200 baud, and bit D5 of `OUT 0FAh` selects the speed while the machine runs. The SOLOS command `SE TA` sets that bit.
- **The modulation is CUTS, not the ACR's FSK.** A Sol tape is `cuts1200` (1200 and 600 Hz) at 1200 baud, one octave below the ACR's 2400 and 1850 Hz. It can also read Kansas City (`kcs300`, 2400 and 1200 Hz). For this reason, the 88-ACR refuses a Sol tape. The tapes chapter tells you more.

When a tape is mounted, the SOLOS commands work:

```
>SA MYPROG 0100 01FF        (save memory to the tape)
>GE MYPROG                  (find it again and load it)
>CA                         (catalog what is on the tape)
```

To write audio that a real Sol can load, you need the right tones, and also the right *shape* and level. A real Sol CUTS modem is a flip-flop that divides a master clock into a square wave. An RC network rounds the wave, and the tape records it at a moderate level. Three properties copy this when the board writes a `.WAV` file. Their defaults come from measurements of a real archived tape:

Property	Default	What it does
`leader`	`3`	Seconds of steady tone before the data. The real tape has 3.05 s. The ACR's leader is longer, 15 s
`trailer`	`2`	Seconds of tone after the data. The real tape has 1.93 s
`rc`	`4000`	The corner of the low-pass filter that rounds the edges, in Hz. It rounds the edges of the square wave as the modem's RC network and the cassette do. CUTS only

`level` (percent of full scale, default `36`) applies to both boards, and the tapes chapter describes it. With these defaults, a CUTS tape from the simulator has its tones on the same clock grid as a real Sol expects, at the level of a real tape. It is made to load on the hardware, not only to read back here.

Add a `vdm1` to this board, and you have the **`sol20`** machine, which starts the real SOLOS operating system.

Interrupts and the clock

Vectored interrupts and real-time clocks.

`virtc`: MITS 88-VI/RTC

Two things on one board.

Vectored interrupts. The board has eight lines, **VI0 to VI7**. You strap a device to a line. When the device interrupts, **level *n* becomes `RST n`**. The processor jumps to `8×n`, and the correct handler runs, with no polling. **VI0 has the highest priority**, and the board enforces it: a lower level cannot interrupt a higher level while the higher level is being handled.

With interrupts, the machine can do other work and does not have to wait in a loop. Every board with an interrupt strap in its properties, such as the serial boards and the floppy controllers, is strapped to one of these lines, to `int`, or to nothing.

A real-time clock, on the same board. It gives a periodic interrupt from the 60 Hz power line, or from the system clock divided down. The `rtc_source` jumper selects which.

The board has one port, at `FE`, and it is **write-only**. You cannot read anything back. Interrupt boards are easy to set up wrong, so read the reference for this board before you strap anything to it.

The `ps2int` machine shows the board in use, with a MITS Programming System II tape **that you supply**. The package has no such tape, so its cassette deck starts empty.

`ss1`: CompuPro System Support 1

A **multifunction** board. On the real board, one block of 16 ports holds a serial channel, an interval timer, two interrupt controllers, a battery-backed **real-time clock and calendar**, and a socket for a math coprocessor. This board models the **clock**, the **serial channel**, the **interval timer**, the **two interrupt controllers** and the **math socket**. The math socket is empty by default.

The clock is an OKI MSM5832. It starts with **the date and time of your computer**, so a guest program that reads it gets the real time. A guest can also **set** it. After that, the clock keeps the setting, and the time

is kept through a RESET, as the battery on the real chip did. The block is at base **`50H`** by default, the CompuPro convention. The `base` property moves it.

The serial channel is a **2651 UART** at `5C` – `5F`. It is a spare serial port. A guest sets its rate, its frame and its RS-232 handshake through the mode and command registers. You connect it with `CONNECT ss1:serial <endpoint>`. Its `baud`, `interrupt` and `connect` are unit settings, which `SHOW ss1` shows.

The interval timer is an **Intel 8253** at `54` – `57`. It has three independent 16-bit counters, which a guest can use as timers or as square-wave and rate generators. The counters count at 2 MHz, and their outputs go to the interrupt controllers. `SHOW ss1` prints a live `timer` line with the mode, count and output of each counter.

The interrupt system is two **Intel 8259A** controllers at `50` – `53`, cascaded as master and slave:

- The master watches the eight vectored interrupt lines of the bus, and drives the processor's interrupt pin.
- The slave collects the sources on the board, which are the three timer outputs and the UART's transmit-ready and receive-ready signals. It sends them to the master.

A guest sets them up in the usual 8259A way, with the ICW and OCW words, and unmasks the sources that it wants. On an interrupt, the controller gives the processor a `CALL` to the vector of the source with the highest priority. `SHOW ss1` prints a live `pic` line with the request, in-service and mask registers of each controller. The master is the interrupt priority encoder of the machine, so do not also add a `virtc`. The two boards would both answer the interrupt acknowledge.

The math socket takes an **AMD 9511A** (Intel 8231A) or an **AMD 9512** (Intel 8232). The `math` property selects the chip: `none` (the default), `9511a` or `9512`. With no chip, the two ports read `FF`, as on a stock board. The chip is at `58` (data) and `59` (command and status). A guest pushes the operands, writes a command and pops the result.

- The **9511A** does 16-bit and 32-bit fixed point, 32-bit floating point, and the functions SQRT, SIN, COS, TAN, ASIN, ACOS, ATAN, LOG, LN, EXP and PWR.
- The **9512** does 32-bit and 64-bit floating point in the format of the proposed IEEE standard: add, subtract, multiply and divide.

Each command takes the number of cycles that the data sheet gives, at 2 MHz. If the guest uses the data port or writes a command before a command ends, the board holds the processor in wait states until the command ends. The chip's END signal goes to slave IR5 and its SVREQ signal goes to slave IR4, so a guest can also wait for an interrupt. `SHOW ss1` prints a live `mathchip` line with the state of the chip, its status and the top of its stack.

```
altairsim> SET ss1 math=9511a
```

In a machine file:

```
[[board]]
type = "ss1"
id   = "ss1"
math = "9511a"
```

The `compupro` machine has one: a stock Altair with the System Support 1 added, and its console still on the 2SIO. The clock is at `5A` (command) and `5B` (data). The board reference gives the digit map, the read and set sequences and the register layouts.

`rtc100`: SciTronics RTC-100

A **battery-backed calendar clock**, and nothing else. SciTronics sold it in 1980 for machines that did not know the date. It has an OKI MSM5832 clock chip, the same chip as on the System Support 1, behind a 6821 parallel interface, on four ports in a row.

It starts with **the date and time of your computer**, so a guest gets the real time. When a guest sets it, the clock keeps that setting through a RESET and a power cycle, as the lithium cell of the real board did. `SHOW <id>` prints a live `time` line with the time on the clock and how far it is from the time of your computer.

The base address is the PORT switch of the board, and it **must be a multiple of 4**. The switch decodes only the top six address bits, and the bottom two bits select one of the four ports. The default is `F0`. A guest reads the clock one BCD digit at a time. It writes the number of the digit to the first port, and reads the digit back **in the top half of the byte**. For this reason, the period drivers for this board all mask and rotate the byte at the end.

It can also **interrupt once a second**, to drive a clock display in the background. Set `interrupt` to `int`, and the board pulls the interrupt line of the machine. `restart` selects the `RST` that it gives the processor (0 to 7, the INT switch of the board). The manual of the board says that 0 and 7 are often used by other things. With `interrupt` set to `none`, the board never interrupts, and it still keeps the correct time.

Host integration

A board that connects the machine to your computer. It is not period hardware.

`hostbridge`: our own board, not a period board

This board never existed. It is the one part of the machine that is not history, and this manual says so.

It moves **files between the guest and your computer**, in both directions. It is **sandboxed**. The guest sees one folder that you choose, and **it cannot leave that folder**. It cannot use `..` or an absolute path to leave it. This is a fixed rule, not a setting.

The default port is `B0`. **The utilities are not in the board.** `R`, `W` and `HDIR` are ordinary CP/M `.COM` programs. They are on a disk, they run at the `A>` prompt, and they use the two ports of the board, as any other CP/M program could. They were assembled *in the machine*, with the machine's own assembler, from 8080 source that ships with the package. You can read their code.

The file-transfer chapter describes this board. It is the fastest way to get your own code into CP/M. XMODEM also works, over an ordinary serial line, as it did on real machines.

PROM programmers

A board that does not run software, but *programs* it into a chip. You load a period EPROM programmer routine and run it. The bytes that it programs go into a socket, and you can save the socket to a file.

`pb1`: SSM PB1

The **SSM PB1** (Solid State Music) is a 2708 and 2716 EPROM programmer. It has a **4K window** in memory (default `D000`) for its two sockets: **U22** holds a 2708 (1K), and **U23** holds a 2716 (2K). It has one **control port** (default `10`). **The default machine has its 2SIO at `10` too**, so move one of the two boards when you add a `pb1`.

Software programs the chip:

1. An `OUT` to the control port arms the board and selects the chip. Write `01` for the 2708, or `02` for the 2716.
2. After that, every byte that the guest **writes into the window** is programmed into the chip.
3. A **read** of the window disarms the board again. The period routines end in this way, and it turns the LED off.

"Programming is a write, and disarming is a read" has two results:

- **A socket starts erased**, with every byte `FF`. **Programming can only clear bits to 0.** It cannot set them, as on a real EPROM. Program a *blank* chip. A second program over the first gives the AND of the two.
- The program is in the board, not on your computer. To keep it, **`SAVE` it to a file** (below). A power cycle reads the sockets again from their mounts, so a program that you did not save is lost.

Programming an EPROM, and making a hex file

You need this board, not only a write to memory, because you want to run the period *software*. The PB1 manual has SSM's programmer routines, one for the 2708 and one for the 2716. Each routine copies bytes from `4000` into the socket. The 2708 routine starts at `0100`. The package does not include these routines.

Enter one from the manual, or `LOAD` it from a hex file of your own. After that, put the data at `4000`, run the routine, and save the result:

```
altairsim> FILL 4000-43FF E5
altairsim> LOAD PROG2708.HEX
altairsim> BREAK F021
altairsim> RUN 100
altairsim> SAVE eprom.hex D000-D3FF
```

- `FILL` puts the 1K that you want to program at `4000`. You can also `LOAD` your own data there.
- `LOAD PROG2708.HEX` loads the 2708 routine.
- `BREAK F021` is needed because SSM's routine ends with a jump to SSM's own system monitor, at `F021`. A machine without that monitor has nothing at `F021`. The breakpoint stops the machine at that jump, after the chip is programmed. The next section shows a machine that has the monitor, and needs no breakpoint.
- `RUN 100` runs the routine. It arms the board and programs the 2708.
- `SAVE` reads the socket window from the bus and writes it to a file. After it is programmed, the socket reads like other memory. `eprom.hex` is an ordinary Intel HEX image of the chip. You can give it to another tool, or `LOAD` it again later.

Your own routine works in the same way. It writes `01` to port `10` (or `02` for a 2716), and then writes the data into the window. After it runs, `SAVE` the window.

With the SSM monitor

SSM wrote the routines for the **SSM 8080 System Monitor**. Each routine ends with a jump to the monitor at `F021`. The manual also has two check routines, which print `P` (pass) or `F` (fail) through the monitor's console-output routine at `F009`. The package has that monitor as the built-in ROM `ssm-8080mon`. Put it in the machine, and the routines run as SSM wrote them.

The monitor has its console on a serial port at `00` and `01`, as an `io4` has by default. Thus this machine has an `io4` as its console, RAM below the window of the `pb1`, and the monitor at `F000`:

```
[machine]
name = "pb1"

[[board]]
type = "8080"
id   = "cpu0"

[[board]]
type = "io4"
id   = "io40"

  [board.unit.a]
  connect = "console"

[[board]]
type   = "pb1"
id     = "pb1"
port   = 10
window = D000

[[board]]
type = "memory"
id   = "mem0"

  [[board.region]]
  type = "ram"
  at   = 0000
  size = "52K"

  [[board.region]]
  type  = "rom"
  at    = F000
  mount = "builtin:ssm-8080mon"
```

Put the data and the routine in memory as before, and then start the monitor:

```
altairsim> FILL 4000-43FF E5
altairsim> LOAD PROG2708.HEX
altairsim> RUN F000
```

The monitor prints its name and its prompt, which is a period:

```
MONITOR V1.0
.
```

Type `G100` and press Enter. `G` is the monitor's command to run a program. The routine programs the chip and then jumps to `F021`, and the monitor prints its name and its prompt again:

```
.G100

MONITOR V1.0
.
```

Press `Ctrl-E` to go back to the `altairsim>` prompt, and `SAVE` the window as before.

A check routine runs in the same way, with `G` and its start address. It prints its result before the monitor prints its name. This is the manual's copy-verify routine, which starts at `0180`, after a good burn:

```
.G180
P
MONITOR V1.0
.
```

The board also has an optional **EPROM area on the board** (U11 to U14 on the real board). These are read-only chips that you mount above `8000` with `[[board.prom]]` (`at` and `mount`). Use them as the *source* when you copy one EPROM to another.

Disks

A disk on an Altair is three things:

- a **controller** board on the bus
- some **drives** on that controller
- a **disk image** in one of those drives

The controller is a board, and it goes in the machine file. The drives are units of the controller. The disk image goes in a drive, from the machine file or with `MOUNT` at the monitor. "Put a disk in the machine file", below, shows the two forms together.

This chapter is about the image, the drive and `MOUNT`. `MOUNT` works in the same way for every controller. The examples use the `dcdd`, the MITS 8″ hard-sector controller, because the CP/M machine in the package boots from it. The `mds` is its 5¼″ relative, and it works in the same way. The Boards chapter describes every controller.

The controllers in this chapter

Board	What it is	Drives	Ports
`dcdd`	**MITS 88-DCDD**: the 8″ hard-sector floppy controller. CP/M booted from it	4 by default, up to 16	08, 09, 0A
`mds`	**88-MDS**: the 5¼″ minidisk. Smaller and slower	4	08, 09, 0A

The two boards use the same ports, so one machine cannot have both. The drives are units of the board: `drive0`, `drive1` and so on.

Put a disk in: `MOUNT`

```
MOUNT <id>[:<unit>] <file> [WP] [CREATE]
```

```
altairsim> MOUNT dsk0:drive1 games.dsk
```

This puts the image `games.dsk` in drive 1 of the controller `dsk0`. A relative path starts from the machine's folder. The machines chapter gives the rule.

The file must already exist. A name that does not exist is a typing mistake, not a new disk. The program tells you so, and does not make a disk that you did not want:

```
altairsim> MOUNT dsk0:drive1 my-scratch.dsk
dsk0: 'my-scratch.dsk': no such file
dsk0: to make a blank one, add CREATE: MOUNT dsk0:drive1 my-scratch.dsk CREATE
```

To make a new disk, see "Make a scratch disk" below.

To take the disk out, type `UNMOUNT`. The guest then sees an empty drive:

```
altairsim> UNMOUNT dsk0:drive1
dsk0:drive1: unmounted (the drive is now empty)
```

Board names are not case-sensitive. When the machine has only one controller, you can leave out the number: `dsk` finds `dsk0`. You must always give the drive, because a controller has more than one.

Put a disk in the machine file

`MOUNT` at the monitor is for the disk that you use *now*. A disk that belongs to a machine goes in the machine file. The examples in the package do this. This is a complete machine file:

```
[machine]
name    = "mine"
base    = "default"
startup = ["RUN FF00"]

[[board]]
id = "dsk0"                    # no type: the controller is already in the machine

  [[board.drive]]
  unit  = 0
  mount = "cpm22b23-56k.dsk"   # relative to this file

  [[board.drive]]
  unit     = 1
  mount    = "games.dsk"
  readonly = true
```

Each `[[board.drive]]` is one `MOUNT` command. The second one is `MOUNT dsk0:drive1 games.dsk WP`:

At the monitor	In the machine file
`dsk0`	`id = "dsk0"` on the `[[board]]`
`drive1`	`unit = 1` in a `[[board.drive]]`
`games.dsk`	`mount = "games.dsk"`
`WP`	`readonly = true`
`CREATE`	`create = true`

All the drives of one controller go below one `[[board]]`. Start a new `[[board.drive]]` for each drive.

Do not write a `type` on that `[[board]]`. The `default` machine already has the controller `dsk0`. With a `type`, the file replaces the controller, and the settings that the base made on it are lost. The configuring chapter gives the rule.

A drive can also have a `media` key, which `MOUNT` does not have. See "How the program finds the format", below.

The file and the monitor do the same thing, at different times. The file puts the disk in the drive when the machine starts. `MOUNT` changes it later.

The name has no folder, so the program looks for the file beside the machine file. For this reason, you can copy the whole folder to another place, and it still boots.

`writeprotect = true` is the same as `readonly = true`. `CONFIG SAVE` saves `readonly`.

Keep your disk safe

Copy the folder before you experiment. The program mounts a disk **read/write**, and every write goes to the file on your computer at once. There is no undo. `SNAPSHOT` does not help, because it saves the state of the machine, not the files on your computer. One wrong `ERA` can destroy the example disk. The folder has everything that the machine needs, and the copy boots from any place.

Read/write is the default because CP/M cannot save your work on a disk that it cannot write.

Go back to the `A>` prompt before you stop

> **Do not `UNMOUNT`, copy or quit right after a file operation. Go back to the `A>` prompt first.**

The CP/M in this package has a **track-buffering BIOS**. When CP/M writes a sector, the BIOS copies it into a buffer in memory that holds one whole track. Nothing goes to the disk yet. The BIOS writes the buffer to the disk when it next reads the console. At the `A>` prompt, CP/M reads the console, so the disk on your computer is then correct.

The BIOS works this way for speed. A track write needs one seek and one turn of the disk, not thirty-two. When the machine waits for a person to type, it has time to spare. The simulator does not cause this. The BIOS of a real machine did the same.

Not every CP/M does this. The author of each BIOS decided it. A *write-through* BIOS puts every sector on the disk before it returns. You cannot tell from the `A>` prompt which kind you have. On a write-through BIOS, the rule costs you nothing. On a buffered BIOS, it decides whether your file is on the disk.

For example, `Ctrl-E` at the `A>` prompt is safe. `Ctrl-E` right after `ED` says that it wrote your file is **not** safe. On the CP/M in this package, the file is then not on the disk.

Write protection: `WP`

`WP` **write-protects the disk**, as the write-protect notch of a real diskette did. The guest can read the disk. The controller refuses every write, and no write reaches the file on your computer.

```
altairsim> MOUNT dsk0:drive2 golden-master.dsk WP
dsk0:drive2: mounted golden-master.dsk (write-protected)
```

In a machine file:

```
[[board]]
id = "dsk0"

  [[board.drive]]
  unit     = 2
  mount    = "golden-master.dsk"
  readonly = true
```

`RO` is also accepted, and it means the same. `RO` is the word for a ROM socket. For a disk, the correct word is `WP`. `SHOW MOUNTS` marks every write-protected disk:

```
altairsim> SHOW MOUNTS
  UNIT         KIND  HOLDS
  dsk0:drive2  disk  golden-master.dsk  (write-protected)
```

Use `WP` only for a disk that the guest reads. The guest is not told that the disk is protected, and it cannot be told:

- **The controller has no bit for it.** The status byte of the 88-DCDD has no bit that means *the notch is covered*. A program cannot tell a protected disk from an ordinary one.
- **CP/M has no error for it.** A CP/M 2.2 BIOS write returns `0` for success or `1` for an error. For this reason, a write to a `WP` disk shows as `Bdos Err On A: Bad Sector`.

The message `Bdos Err On A: R/O` comes from a different thing. It is CP/M's own read-only flag, which `STAT A:=R/O` sets and a warm boot clears. The flag is in CP/M's memory, not on the disk. A `WP` disk never causes it.

Look for this line in `SHOW MOUNTS`. If your computer does not let the program write the file, because of its permissions, the disk still mounts, write-protected. The program tells you when it mounts the disk, and `SHOW MOUNTS` says that you did not ask for it:

```
dsk0:drive2  disk  master.dsk  (write-protected -- THE HOST WON'T LET US WRITE IT; you did not ask for this)
```

CP/M will fail every write to that disk, and you think that it is writable.

Make a scratch disk

`CREATE` gives you a blank medium, and a format program in the guest makes it a disk.

1. At the monitor, make the file and put it in a drive:

   ```
   altairsim> MOUNT dsk0:drive1 my-scratch.dsk CREATE
   dsk0:drive1: created my-scratch.dsk (empty)
   dsk0:drive1: mounted my-scratch.dsk
   ```

   The file has zero bytes. It is an unformatted disk, and CP/M cannot read it yet.

2. Boot CP/M, and format the disk with **`AFORMAT`**, the Altair Disk Format Utility by Martin Eberhard. It is on the CP/M disk in this package:

   ```
   A>AFORMAT B:

   Put disk in B
   Ready (Y/N)?Y
   Formatting an 8" Disk.....................................................................
   Format another (Y/N)?N
   Put CP/M disk in A:
   Ready (Y/N)?Y
   A>
   ```

   Each dot is a track. `AFORMAT` formats 77 tracks, so the file is now a 337,568-byte 8″ floppy.

3. Type `DIR B:`. CP/M answers `No file`, and `B:` is ready for `PIP` and `SAVE`.

In a machine file, `create = true` does step 1:

```
[[board]]
id = "dsk0"

  [[board.drive]]
  unit   = 1
  mount  = "my-scratch.dsk"
  create = true
```

The program makes the file only when the file is not there. For this reason, the same machine file loads the first time and every time after it, and the disk keeps its contents.

`CREATE` works only on a controller that can grow its disk. The 88-HDSK "Datakeeper" hard disk has a fixed size, so it cannot start from an empty file. It refuses `CREATE`, and removes the empty file:

```
h0: 88-HDSK has a fixed geometry and cannot make a blank platter -- supply an existing 4988928-byte image (406 cyl x 2 sides x 24 sectors x 256)
```

For that disk, you supply a full-size image.

How the program finds the format

You do not tell the program what kind of disk you mounted. It **looks at the number of bytes in the file**:

Format	Tracks	Sectors	Bytes/sector	File size
`8in`	77	32	137	**337,568**
`minidisk`	35	16	137	**76,720**
`fdc8mb`	2048	32	137	**8,978,432**

A file of 337,568 bytes is an 8″ floppy, and it cannot be anything else. The `8in` and `fdc8mb` formats belong to the `dcdd`, and `minidisk` belongs to the `mds`. The program chooses only from the formats of the board.

A file can be a little bigger than its format. XMODEM adds bytes to fill a 128-byte block, and most images in circulation were sent by XMODEM at some time. The CP/M disk in this package is 337,664 bytes, and the minidisk images that you find are 76,800 bytes. The program allows for this.

A file whose size matches no format mounts UNFORMATTED, at the largest format of the board: `fdc8mb` on a `dcdd`, and `minidisk` on an `mds`. A blank file from `CREATE` is one of these. The file grows as

the guest writes it.

`media` sets the format yourself. Use it on the drive in a machine file when the size cannot decide. Examples are an image that is cut short, a format that you made up, or a blank disk for a format program that formats every track that the drive can reach:

```
[[board.drive]]
unit   = 1
mount  = "scratch.dsk"
create = true
media  = "8in"        # a blank 77-track floppy, not a blank 8 MB disk
```

`media` is a key in the machine file only. You cannot set it with `MOUNT`. **For every other disk, leave `media` out.** A wrong `media` gives disk errors in the guest, and the troubleshooting chapter tells you why.

The package has an 8″ floppy image, the CP/M disk. You supply any `minidisk` or `fdc8mb` image.

How an 8 MB disk works

`fdc8mb` is a disk with 2048 tracks on a controller that MITS designed for 77 tracks. It works with **the stock board and the stock PROM**. The controller steps the head and moves bytes, and it never asks how big the disk is. The BIOS knows where track 1500 is.

For this reason, **the format belongs to each drive, not to the controller**. One board can have drives of different sizes. The period 8 MB CP/M BIOSes expect an 8 MB disk on A: and B:, and 77-track floppies on C: and D:, so that you can `PIP` between them.

You supply both images. The lines below show the form of the commands:

```
altairsim> MOUNT dsk0:drive0 big.dsk
altairsim> MOUNT dsk0:drive2 floppy.dsk
altairsim> SHOW dsk0
```

In a machine file:

```
[[board]]
id = "dsk0"

  [[board.drive]]
  unit  = 0
  mount = "big.dsk"

  [[board.drive]]
  unit  = 2
  mount = "floppy.dsk"
```

ImageDisk files: `.imd`

Much archived 8″ and 5¼″ software is in the **ImageDisk** format. A controller reads raw sectors, so the program does not mount an `.imd` directly. When you `MOUNT` a file whose name ends in `.imd`, the program converts it to a raw image beside it, with the name `.dsk`, and mounts the `.dsk`:

```
altairsim> MOUNT dsk0:drive0 cpm.imd
dsk0: converted cpm.imd -> cpm.dsk
dsk0:   IMD: CP/M 2.2 system disk
dsk0:   337568 bytes
dsk0:drive0: mounted cpm.dsk
```

Read the comment line of the `.imd` in the output, and check that it is the disk that you expected.

The program never writes over a `.dsk` that is already there. If `cpm.dsk` exists, the program tells you to remove it or to mount it directly. For this reason, a second mount cannot lose the changes that you made. After the conversion, the `.dsk` is the disk.

A disk on a TNFS server

A disk image can be on a **TNFS server**, the network file system of the FujiNet project:

```
altairsim> MOUNT dsk0:drive0 tnfs://fileserver/cpm/games.dsk
altairsim> MOUNT dsk0:drive1 tnfs://fileserver:16384/scratch.dsk WP
```

The form is `tnfs://<host>[:<port>]/<path>`. The default port is **16384**.

In a machine file:

```
[[board]]
id = "dsk0"

  [[board.drive]]
  unit  = 0
  mount = "tnfs://fileserver/cpm/games.dsk"

  [[board.drive]]
  unit     = 1
  mount    = "tnfs://fileserver:16384/scratch.dsk"
  readonly = true
```

A `tnfs://` name does not start from the machine's folder, because it names its server.

The program gets the whole image **one time, when you mount it**. After that, the disk works like a local disk. The program writes your changes back to the server when the guest flushes, and when you `UNMOUNT` the disk. `WP` works as on a local disk. If the server does not let the program write the file, the disk mounts write-protected, as a local file does.

If the server stops while you use the disk, the program tells you. It prints a line that says that it keeps the changes in memory only. The guest continues to run, and the program continues to try. When the server comes back, the program says so, and saves the changes. If you quit before that, the changes are lost.

TNFS carries only **single-file images**: a floppy, a minidisk, an 8 MB `fdc8mb` disk and a cassette tape. The card images of the CompactFlash and SD boards have a second file (`.img` with `.geo`), and you cannot mount them over TNFS.

Look at what is in the machine

`SHOW` on the controller lists its drives and the disk in each one. `BOARDS` lists every board and the addresses that it decodes:

```
altairsim> SHOW dsk0
altairsim> BOARDS
```

These two commands answer most questions about a machine that does not boot. The most common answer is that the disk is in the wrong drive.

Tapes

Before the floppy disk, there was the **cassette recorder**. It was an ordinary home audio cassette deck, with a microphone jack and an earphone jack. MITS sold a board that changed bytes into a sound that the deck could record, and changed the sound back into bytes. That board is the **88-ACR**, and it is in this simulator.

This chapter starts with the most important use of the ACR: loading Altair 4K BASIC 3.1 as people loaded it in 1976. After that, it describes how to use a tape: mount it, move it, play it at its real speed, use audio `.WAV` files, and record.

The **Sol-20** has its own cassette interface, with two decks and motor control that the guest can use. Everything in this chapter also works on the Sol's decks, with the unit names `sol0:tape1` and `sol0:tape2`. The `sol` section of the Boards chapter gives the differences.

Load Altair 4K BASIC 3.1 by hand

An Altair owner did this every time that they wanted to use BASIC. There was nothing in ROM and no disk. Type the commands in the folder where you unzipped the package.

1. Start the built-in `basic4k` machine. It has an 88-ACR, and its sense switches are set for a cassette load:

   ```
   $ altairsim basic4k
   ```

2. Put the cassette in:

   ```
   altairsim> MOUNT acr0:tape "examples/basic4k/4K BASIC Ver 3-1.tap"
   ```

3. Enter the bootstrap:

   ```
   altairsim> LOAD "examples/basic4k/LDR4K31.HEX"
   ```

   The bootstrap is 20 bytes. On a real Altair, you entered it by hand on the front-panel switches, from the listing that MITS printed in its manual. You set eight switches for each byte, and pressed DEPOSIT. `LOAD` puts the same 20 bytes in memory.

4. Run the bootstrap from address zero:

   ```
   altairsim> RUN 0
   ```

The bootstrap reads the tape, BASIC loads into memory, and BASIC starts.

5. Answer the three questions. Press Return for the first two, and type `Y` for the third:

```
MEMORY SIZE?
TERMINAL WIDTH?
WANT SIN? Y

742 BYTES FREE

ALTAIR BASIC VERSION 3.1
[FOUR-K VERSION]

OK
```

   An empty answer means "all of the memory" and "the default width". `WANT SIN?` asks whether to use some of your 4K for trigonometry.

You are in Altair BASIC, the first product that Microsoft sold. The *Worked examples* chapter does the same with one command, `altairsim examples/basic4k/basic4k.toml`.

If a tape load stops, and the tape is mounted, check the sense switches first. The bootstrap reads them to find the device to load from. The `basic4k` machine sets `sense = 0x80` on the front panel: A15 on, for a cassette load, and all other switches off. With a different setting, BASIC waits for bytes from a device that is not there. The `fp` section of the Boards chapter describes the switches.

The 88-ACR

`acr` is the **MITS 88-ACR**, an Audio Cassette Record and playback interface. It is **an 88-SIO channel B with an FSK modem connected to it**, as MITS built it. The guest uses an ordinary serial port, and the modem changes the bits into tones.

- The default port is **06**. `06` is status and control, and `07` is data.
- It runs at **300 baud**, the speed of the tape. You cannot change it.
- It has one unit, **`tape`**, because a cassette recorder has one place for a tape.

You cannot `CONNECT` the ACR. Its line goes to the modem on the board, and the modem goes to a cassette deck. There is no connector for a terminal on the back of an 88-ACR. The serial chapter describes the boards that take endpoints.

You put the tape in by hand. An 88-ACR cannot start, stop or rewind the tape, and the machine does not know that a tape is there. For this reason, a machine file has no key for the tape. The tape is not hardware.

You put it in with `MOUNT`, and a `startup` command can type the `MOUNT` for you, as `examples/basic4k/basic4k.toml` does.

Use a tape

Put a tape in

```
altairsim> MOUNT acr0:tape "examples/basic4k/4K BASIC Ver 3-1.tap"
```

The ACR has one unit, so you can leave out the unit and the number:

```
altairsim> MOUNT ACR tape.bin
```

A Sol has two decks, so you must name the deck.

`mode = play | record`

```
altairsim> SET acr0:tape mode=record
```

`play` loads from the file, and `record` saves to it. A tape has one head, and it moves in one direction at a time, so `mode` has two values only.

The tape counter

`SHOW MOUNTS` and `SHOW <id>` give the position of the head:

```
altairsim> SHOW MOUNTS
  UNIT       KIND  HOLDS
  acr0:tape  tape  4K BASIC Ver 3-1.tap  00:00 / 02:28 (0%)  1/4439 bytes
```

The counter gives minutes and seconds into the tape, a percentage, and a byte count. For a `.WAV` file, the time is the recording's own time, with the leader and the silent gaps between programs, as on a real cassette counter. For this reason, a program that a manual finds "so many seconds from the start of the tape" is at that second here. A byte `.TAP` file has no audio, so the program calculates its time from the baud rate.

When a tape plays at its real speed (`rate = real`, below), the counter **counts up on the console while the tape loads**. Turn it off for a machine whose guest writes to the same terminal:

```
altairsim> MOUNT acr0:tape "tape.wav" counter=off
```

`SET acr0:tape counter=off` also works. When the counter is off, `SHOW` still gives the position.

`WIND` and `REWIND`: move the head

```
WIND <id>:<unit> <mm:ss | START | END>
REWIND <id>:<unit>
```

`WIND` (`WI`) moves the head to a time: `mm:ss`, a number of seconds, `START` or `END`. On a cassette with several programs, read the counter or a manual for the start of the next program, and wind to that time:

```
altairsim> WIND acr0:tape 0:05
acr0:tape: wound to 00:05 / 02:33 (3%) -- 4K BASIC Ver 3-1.wav (1 of 4439 bytes)
```

`REWIND` (`REW`) is the same as `WIND … START`. **Use it to load the same tape a second time.** After the guest reads a tape, the head is at the end, and a second read gets nothing.

`stop`: stop the tape at a time

On a tape with several programs, one program runs into the next. The `stop` mark is like pressing the **STOP button of the recorder at a counter mark**. Set it with `MOUNT`, or with `SET` at any time:

```
altairsim> MOUNT acr0:tape "tape.wav" stop=2:05
altairsim> SET acr0:tape stop=2:05
```

The line goes silent at 2:05, as at the end of the tape, and the loader does not read into the next program. To continue, move the mark forward or clear it, and play again:

```
altairsim> SET acr0:tape stop=5:30      (the next program ends here)
altairsim> SET acr0:tape stop=off       (play to the end of the tape)
```

The mark is `off` until you set it. `SHOW` shows a mark as `stop @ 02:05`. The mark stops **playback only**, and a recording writes past it. If you wind the head past a mark, the tape stays stopped there, and `SHOW` says why.

Speed: `rate = full | real`

The tape has its own clock, and by default it runs as fast as possible. With `rate = full`, the guest gets each byte as soon as it is ready to read the next one. A cassette that took a real Altair about **110 seconds** loads in **about one second**.

```
altairsim> SET acr0:tape rate=real
```

`rate = real` plays the tape at the tape's baud rate, 300 or 1200, and a load takes as long as it took on the real machine. Use it when you want to show the wait, for example in a demonstration. It applies to playback only. A recording takes as long as the guest takes.

`rate` is not the processor clock. The tape and the processor had separate crystals on the real hardware, and they are separate here. `SET cpu0 clock_hz=2000000` gives the *processor* the speed of the period, so that a game plays at its real speed. It does not change the speed of the tape. With `rate = full`, the guest sees no difference at any processor speed, because a polled loader only asks for the next byte, and the byte is always ready.

Audio tapes: `.WAV`

Most surviving Altair and Sol cassettes are **audio**. Somebody played a cassette into a sound card, and saved a `.WAV` file. You can mount one on either board.

`examples/basic4k/` has 4K BASIC on an audio tape, `4K BASIC Ver 3-1.wav`. It holds the program of the `.tap`, encoded in the audio format of the 88-ACR. The machine file `basic4k-wav.toml` mounts it and boots. In that folder, type:

```
altairsim> MOUNT acr0:tape "4K BASIC Ver 3-1.wav"
acr0:tape: mounted 4K BASIC Ver 3-1.wav
4K BASIC Ver 3-1.wav: fsk300, 4439 bytes, 0 framing errors (100.0% of frames intact)
```

`examples/basic1/` has Altair BASIC 1.0 on an audio tape too, `BASIC Ver 1-0.wav`. The machine file `basic1-wav.toml` mounts it.

The program decodes the recording **one time, when you mount it**, and never while the machine runs. After that, a `.WAV` works as a `.TAP` does. The counter gives the real minutes and seconds of the recording, and the guest sees no difference.

Read the second line of the mount output. It gives the number of framing errors. A tape that decodes at 60% is noise, not a program, and the program refuses a decode below 90%. The percentage counts only

the frames whose start and stop bits were in the correct place. A worn recording can keep its framing and still give wrong bytes. If a tape mounts well and the program still does not run, the recording is probably worn.

The start of the file decides the format, never its name. A `.TAP` file that somebody renamed `.WAV` is still read as bytes. A recording renamed `.TAP` is still decoded as audio.

`extract`: one `.TAP` file for each program

A cassette often holds several programs, with a few seconds of silence between them. **`extract` writes each program to its own `.TAP` file**, so that you can mount or load one program at a time:

```
altairsim> MOUNT acr0:tape "games.wav" extract
acr0:tape: mounted games.wav
  games-1.tap  2048 bytes
  games-2.tap  3120 bytes
2 programs extracted
```

The files go **beside the WAV**, numbered from 1. A tape with one program gives `games.tap`, with no number. `extract=<base>` sets the names, for example `extract=disk1` gives `disk1-1.tap`. `extract` reads the tape and writes the files. Nothing in the machine changes.

To split a WAV that is already in the deck, use the `EXTRACT` command:

```
altairsim> EXTRACT acr0:tape
```

You can extract only a `.WAV`. The program splits at one second or more of silence. That is much longer than any gap inside a program, so no program is cut in half.

A board refuses audio that it cannot hear

`examples/basic4k/` also has 4K BASIC in a modulation that the 88-ACR *cannot* read, `4K BASIC (Kansas City).wav`. The board refuses it:

```
altairsim> MOUNT acr0:tape "4K BASIC (Kansas City).wav"
acr0: 4K BASIC (Kansas City).wav: this board's modem cannot hear that tape -- it carries 2400 Hz / 1200 Hz,
and this board reads fsk300
```

Cassette audio of the period uses different tones:

- The 88-ACR uses **2400/1850 Hz FSK**.
- Many archive tapes are **Kansas City**, **2400/1200 Hz**.
- The Sol's CUTS tapes are one octave lower, **1200/600 Hz**, at 1200 baud.

The decoder of the program could read all of them, but a real 88-ACR could not. Its decoder is a PLL centered at 2125 Hz, with a range of about ±100 Hz. A 1200 Hz tone is far outside that range, so the real board reads **nothing** from that tape. If the program decoded the tape anyway, it would give the guest data that no 88-ACR could produce. For this reason, the board tells you what the tape is, and you use a machine that can read it.

The frequencies in the message are a measurement. On a worn recording, a value can be one octave wrong. The message means "*this is not my format, and this is about what is there*".

Record a tape

Set the recorder to `record`, and run the guest. When the tape stops, the program writes the recording to the file:

```
altairsim> SET acr0:tape mode=record
altairsim> RUN 0                        (the guest records)
altairsim> SET acr0:tape mode=play      (the program writes the file here)
```

The tape stops when you set `mode=play`, `UNMOUNT`, `WIND`, `REWIND` or `QUIT`. On a Sol, the guest can also stop the motor. The program writes the whole file again each time.

A recording writes over the file. A mounted `.WAV` is recorded in the format and at the sample rate that the program found when it decoded the file. Record on a *copy* if you want to keep the original.

A blank tape: `MOUNT … CREATE`

```
altairsim> MOUNT acr0:tape new.wav CREATE mode=record
altairsim> MOUNT acr0:tape new.tap CREATE mode=record
```

A blank file has nothing to examine, so its **name** decides what it becomes. A `.wav` name makes a blank **audio** tape. Any other name makes a blank **byte** tape. The mount line tells you which one you got:

```
new.wav: blank fsk300 tape, ready to record
new.tap: blank raw byte tape
```

Give `mode=record`, because the deck starts in PLAY. The guest can then write to the tape, for example with `CSAVE` in BASIC.

A file that already exists is different. Its start decides, never its name. A `.wav` file that is not really a WAV is read as bytes, and a recording then puts bytes in it, not audio. Nothing can play it. If you wanted audio and the mount line says `raw`, this is what happened.

The leader, the tone and the level

A byte image holds no times, so the program adds the leader that a real tape needs when it writes audio. These properties of the tape unit shape the audio:

Property	88-ACR default	What it does
`leader`	`15`	Seconds of steady tone before the data. The MITS manual asks for about 15 s
`trailer`	`5`	Seconds of tone after the data. The MITS manual asks for 5 s between programs
`waveform`	`square`	`square` is the tone of the real modem, fuller and louder. `sine` is smoother and quieter. Both decode to the same bytes
`level`	`36`	The recording level, in percent of full scale. This is the level of a real tape. A much higher level overloads the input of a real deck

Set `leader` or `trailer` to `0` to make the file as small as possible. The archive `.wav` files are cut in this way, and for this reason they do not load on real hardware. Even at `0`, the program writes sixteen bit times of tone at each end, because the loader finds a start bit by its edge.

The Sol's decks have shorter defaults, and one more property, `rc`. The `sol` section of the Boards chapter gives them.

A tape with several files comes back as one continuous recording. The bytes do not mark where one file ends, so the gaps between programs are not written back.

None of this applies to a `.TAP` file.

`format`: choose the format yourself

Each tape unit has a `format` property. The default, `auto`, is almost always correct.

```
altairsim> SET acr0:tape format=raw
altairsim> SHOW acr0
```

Value	What it does
`auto`	Decode a WAV file as audio. Read any other file as bytes
`raw`	Read the bytes of the file, **also for a WAV**. Use it to look at a tape that decodes badly
a modulation	Decode with this modulation: `fsk300` on the ACR, `cuts1200` or `kcs300` on the Sol

`format` selects how the program *reads* the file. It never changes the hardware. If you tell an 88-ACR to decode `cuts1200`, it refuses. The read-only `detected` property gives the format of the mounted tape.

`format` also selects the modulation of a **blank** tape from `MOUNT … CREATE`. A `.wav` name records with the board's own modem, and `format` selects a different one by name.

`format` takes effect at the **next** `MOUNT`, because the program decodes a tape when you put it in. `SET` names the unit, and `SHOW` names the board. There is no `SHOW <id>:<unit>`.

Serial ports, sockets and telnet

The Altair had no screen. It had a serial board, and you connected a device to it: a Teletype, a video terminal, a modem or a paper-tape reader. The board did not know which device it was. It moved characters.

`altairsim` works in the same way. **Every board that moves characters has one or more UNITS, and you can CONNECT each unit to an ENDPOINT.** The unit is the connector on the back of the board. The endpoint is what you connect to it.

```
altairsim> CONNECT sio0:a socket:2323
altairsim> DISCONNECT sio0:a
```

The same in a machine file

`CONNECT` connects a unit now. A machine file makes the same connection each time that the machine starts. `CONNECT <id>:<unit> <endpoint>` becomes `connect = "<endpoint>"` in the `[board.unit.<unit>]` table of the board that has that `id`.

```
altairsim> CONNECT sio0:a telnet:2323
```

In a machine file:

```
[machine]
name = "cpm-telnet"
base = "default"

[[board]]
id = "sio0"                    # no type: change the board that is there

  [board.unit.a]
  connect = "telnet:2323"
```

This is a complete machine file. The `default` machine already has the board `sio0`. The file names that board by its `id`, and gives only the key that changes. All the other settings of the board stay.

Do not write a `type` on that `[[board]]`. With a `type`, the file replaces the board, and the settings that the base made on it are lost. The configuring chapter gives the rule.

The endpoint is always one string in quotes, with its options and its tap.

You can get the same result in two other ways:

- Type the `CONNECT` command at the monitor, and then type `CONFIG SAVE mine.toml`. The saved file is a complete machine with no `base`. It lists every board and every setting.
- Put the command in the `startup` list: `startup = ["CONNECT sio0:a telnet:2323", "RUN FF00"]`.

In this chapter, the words "In a machine file:" introduce the machine-file form of the command above them. Each sample shows only the tables that change.

The endpoints

This is the complete list.

Endpoint	What it is
`console`	the terminal where you started the program: your keyboard and your screen
`null`	nothing. Writes go nowhere, and a read never gets a byte
`loopback`	the unit itself. What the guest writes comes back as a read
`socket:PORT`	**listens** on that TCP port, and passes the bytes unchanged. Add `?banner` to greet each caller
`socket:HOST:PORT`	**calls** that host and port, and passes the bytes unchanged
`telnet:PORT`	**listens** like `socket:PORT`, and uses the **Telnet protocol**, so that a `telnet` client works well. Use it when a **person** connects. It greets each caller, and `?banner=off` stops that
`telnet:HOST:PORT`	**calls** like `socket:HOST:PORT`, as a telnet client
`serial:DEVICE`	a real serial port on your computer
`pty`	a pseudo-terminal that a terminal program opens as a serial port. Only on macOS and Linux. `pty:LINK` selects the name
`in:PATH`	a file that the board reads: a **paper-tape reader**
`out:PATH`	a file that the board writes: a **paper-tape punch**
`in:PATH,out:PATH`	a reader and a punch on one line, with two files
`terminal`	a terminal in its own window: a VT100, ADM-3A, VT52 or H19
`printer:QUEUE`	a print queue on your computer. Only on a build that has host printing
`scripted`	a terminal that a program types into, for the MCP tools and the tests

You can add two things to any endpoint. `|FILE` writes every byte on the line to a log file, and `|socket:PORT` lets a second person watch the line and type on it (`|pty` does the same for a terminal program, on macOS and Linux, and `|serial:DEVICE` does the same for a terminal on a serial port). See "Tap a line to a log file" and "Mirror a line", below.

`null` is not an error

A unit that is not connected is `null`. **This is a correct state, not a fault.** A 6850 with no cable is always ready to send. A program that writes to it runs correctly, and nobody receives the output. A read never completes, because nothing sends. A real board with no cable does the same.

The colon makes the difference

`socket:2323` **listens**. `socket:localhost:2323` **calls out**. A port alone is a port that you own. A host with a port is a place that you go to.

Only one unit can hold the console

The console is **your keyboard**, and you have one. When you connect a second unit to `console`, **it takes the console, and says which unit it took it from**:

```
altairsim> CONNECT sio1:a console
console taken from sio0:a
sio1:a: connected to console
```

This is not an error. If two boards read one keyboard, each one would get about half the characters. `SHOW CONSOLE` shows which unit has the console, and the console settings.

Use telnet to connect to the guest

Connect a unit to a `telnet:` port, and the guest has a serial port with a terminal on it:

```
altairsim> CONNECT sio0:a telnet:2323
altairsim> RUN
```

In a machine file:

```
[[board]]
id = "sio0"

  [board.unit.a]
  connect = "telnet:2323"
```

In another terminal on your computer, type:

```
$ telnet localhost 2323
Connected to AltairSim X.Y.Z (sio0:a) on port 2323
```

The console of the guest is now that window. Your first terminal still has the monitor and `Ctrl-E`.

Unit `a` is the console port of the `default` machine. To keep the console where it is and give the guest a second terminal, connect unit `b`: write `sio0:b` at the monitor, or `[board.unit.b]` in the machine file.

`socket:` or `telnet:`

Both raise carrier when a caller connects, and drop it when the caller leaves. The guest cannot tell them apart. They differ in what they send to the far end:

	`socket:`	`telnet:`
Use it for	another program or another machine	a **person** with `telnet` or `nc`
What goes on the line	the bytes of the guest, and nothing else	the bytes of the guest, and the Telnet settings for echo and single keys
What the person sees	each key two times, and a whole line sent at Enter	each key one time, sent when it is pressed
Greeting	none, unless you add `?banner`	one line with the machine, the unit and the port

Use `telnet:` when a person types to the guest. Use `socket:` when a program is at the far end: one `altairsim` connected to another, a file transfer, or a tool that uses raw bytes.

A `telnet` client on a raw `socket:` shows each key two times, because the client and the guest both echo it. Enter sends a whole line, with a line feed in place of the carriage return. You cannot fix this from the keyboard. The Telnet protocol sets up the two ends to agree.

The connect banner

A `telnet:PORT` line greets each person who connects, before the guest sends anything. This is the `Connected to ...` line above. It names the build, the unit and the port, so that you know that you reached the correct machine. Only the caller sees it, and nothing goes to the board. To turn it off, add `?banner=off`:

```
altairsim> CONNECT sio0:a telnet:2323?banner=off
```

A `socket:PORT` line has no banner, because a program at the far end would read the banner as data. To greet a person on a raw socket, add `?banner`: `CONNECT sio0:a socket:2323?banner`. A banner is only for

a port that listens. On a line that calls out, the program refuses `?banner`.

A terminal in its own window

A telnet client must be installed and pointed at the correct port. It changes any byte that looks like a telnet command, which can break cursor sequences and file transfers. A modern terminal also emulates a modern terminal, not the ADM-3A or VT52 that the software of the period expects.

`terminal` is a terminal that the program draws itself, in its own window:

```
altairsim> CONNECT sio0:a terminal
altairsim> RUN
```

The console of the guest is now that window. The monitor and `Ctrl-E` stay in the terminal that you started from. The window opens when you connect the line. It answers the reports that a program of the period asks for, such as `ESC[6n`.

`terminal` alone is a VT100 in an 80×24 window. Give options after a `?`, joined by `&`:

```
altairsim> CONNECT sio0:a terminal?emulation=adm3a
altairsim> CONNECT sio0:a terminal?emulation=vt52&size=80x24
altairsim> CONNECT sio0:a "terminal?emulation=h19&size=132x24"
altairsim> CONNECT sio0:a terminal?emulation=vt100&phosphor=amber&width=1100
```

In a machine file:

```
[[board]]
id = "sio0"

  [board.unit.a]
  connect = "terminal?emulation=vt52&size=80x24"
```

The options are part of the string. A machine file always has quotes around the endpoint, so the `&` needs no special quotes there.

- `emulation` is `vt100` (the default, and `ansi` is the same), `adm3a` (the Lear Siegler ADM-3A, the usual CP/M terminal), `vt52`, or `h19` (the Heath/Zenith H19, a VT52 with an ANSI mode).
- `size` is *columns*×*rows*. The default is `80x24`.
- `phosphor` is the color of the tube: `green` (the default) or `amber`.
- `width` is the width of the window when it opens, in pixels. Without it, the window is about half as wide as the screen.

The text uses the **DEC VT220** character set. It goes well with `[display] crt = true`, which the configuring chapter describes. With `crt = true`, a `width` opens the window at that size.

The window has its own settings for changing bytes, under `[terminal]`. See "The built-in terminal has these too", below.

Call another computer

```
altairsim> CONNECT sio0:b socket:bbs.example.com:23
```

The program opens the connection. The guest sees a serial line with carrier, and it can run a terminal program of the period over it.

A real serial port

```
altairsim> CONNECT sio0:a serial:/dev/tty.usbserial-A600K1XY
altairsim> CONNECT sio0:a serial:COM3
```

The second form is for Windows. The bytes go out of a real UART, down a real cable, to the device at the other end. The modem control lines (DCD, CTS and RTS) are connected too.

`SHOW SERIAL` lists the serial ports on your computer. For each port, it also shows the unit that holds it:

```
altairsim> SHOW SERIAL
serial  (host serial ports for CONNECT <id>:<u> serial:<device>)

  DEVICE                          IN USE BY
  /dev/cu.Bluetooth-Incoming-Port
  /dev/cu.debug-console
  /dev/cu.usbmodem1121101
  /dev/cu.usbserial-AB0NW409
  /dev/cu.usbserial-AL009KFH

  Only names that look like a serial port are listed. Another device path
  still opens with serial:<path>.
```

The list has only the names that look like a serial port. A device with a different name still opens when you give its full path. If you give a wrong device name, `CONNECT` also lists the ports. A cable can appear under a name that is one character different from what you expected, and the list shows you the correct name.

The board sets the frame on the line

The program opens the port at 9600 8N1. **The board then sets the port again at once**, because only the board knows what frame it sends. To send 300 baud, 7 bits and even parity, set it on the **board**, where an operator in 1975 set it:

- On an **88-SIO** or an **88-ACR**, jumpers set the word format. The board's `baud`, `data_bits`, `stop_bits` and `parity` properties are the frame on the line.
- On an **88-2SIO**, the baud rate is a jumper (`baud`), but the **guest writes the word format to a register** of the 6850. For this reason, the 2SIO has no `data_bits` property. The frame on the line is what the guest last set. A guest that selects 7E1 sets the cable to 7E1.

`data_bits` is a frame, not a mask. `data_bits=7` puts seven data bits in each character on the line. It does not clear bit 7 of the byte that the guest wrote. Do not use it to fix a prompt on your screen. Use the console settings (below).

Give the machine its real crystal before you transfer a file

```
altairsim> SET cpu0 clock_hz=2000000
```

A guest has no clock. It counts instructions. At full speed, its "three-second" timeout passes in a few milliseconds, and it decides that your sender is dead. Full speed is correct for a machine that talks only to itself. **A machine that talks to the outside world needs the real crystal.** The troubleshooting chapter tells you more.

The port, the baud rate and the crystal in a machine file

The three commands are:

```
altairsim> SET cpu0 clock_hz=2000000
altairsim> SET sio0:a baud=9600
altairsim> CONNECT sio0:a serial:/dev/tty.usbserial-A600K1XY
```

In a machine file:

```
[[board]]
id       = "cpu0"
clock_hz = 2000000             # SET cpu0 clock_hz=2000000

[[board]]
id = "sio0"

  [board.unit.a]
  baud    = 9600               # SET sio0:a baud=9600
  connect = "serial:/dev/tty.usbserial-A600K1XY"
```

A `SET` on a board is a key in the `[[board]]` of that board. A `SET` on a unit is a key in the table of that unit.

A pseudo-terminal for a terminal program

On macOS and Linux, a line can go to a **pseudo-terminal**. A terminal program, such as `screen`, `minicom` or `cu`, opens a pseudo-terminal as it opens a serial port. No cable and no port number are necessary. A file transfer program can use the line also, because each byte goes through unchanged.

```
altairsim> CONNECT sio0:b pty
sio0:b: connected to pty
pty: /tmp/altairsim0 (/dev/ttys004)
```

The program prints the name of the line. The first line is `/tmp/altairsim0`, and the next one is `/tmp/altairsim1`. Type `screen /tmp/altairsim0` in a second window to use the line. The name is a link to the device in the brackets. The program removes the link when the line disconnects, and when the program stops.

To select the name, give a path after `pty:`. The name is then the same each time. A path that is not absolute is relative to the folder of the machine file.

```
altairsim> CONNECT sio0:b pty:/tmp/altair-line
sio0:b: connected to pty:/tmp/altair-line
pty: /tmp/altair-line (/dev/ttys004)
```

In a machine file:

```
[[board]]
id = "sio0"

  [board.unit.b]
  connect = "pty:/tmp/altair-line"
```

When the machine file starts the machine, the program prints the same `pty:` line, with the name of the unit before it.

- **When no program has the name open, the line discards what the guest sends.** The guest does not wait.
- **A program that has the name open is the carrier.** The DCD and DSR pins of the line are on while a program has the name open, and off when it closes the name. On a 2SIO, the guest sees DCD only after `SET sio0:b dcd=wired`.
- **A program that reads slowly makes the guest wait.** The guest loses no bytes.
- **`pty:` replaces a link that is at the path.** If the path is a file that is not a link, the program refuses, and does not replace the file.
- **Windows has no pseudo-terminal, and refuses `pty`.** Use `socket:PORT` or `telnet:PORT`.

To let a second person watch a line that goes to a different endpoint, use `|pty`. See "A mirror on a pseudo-terminal", below.

A paper-tape reader and punch: `in:` and `out:`

A **paper-tape station** was connected to a serial or parallel line. The program makes one from two files on your computer:

```
altairsim> CONNECT lpt0:prn out:printout.txt              # a punch: save what the board sends
altairsim> CONNECT 4pio0:ja in:reader.tap                 # a reader: give a file to the board
altairsim> CONNECT 4pio0:ja in:reader.tap,out:punch.tap   # both, on one line
```

In a machine file, for a machine that has the board `4pio0`:

```
[[board]]
id = "4pio0"

  [board.unit.ja]
  connect = "in:reader.tap,out:punch.tap"
```

The name of the table is the name of the unit: `ja` here, `a` on a 2SIO.

`in:` is a **reader**. It gives the bytes of the file to the board one at a time, from the start. When the file ends, the line goes **quiet**, with no error and no end-of-file byte, as a reader with no more tape does. If the file does not exist, `CONNECT` refuses, and names the path.

`out:` is a **punch**. The board's output goes to the file. If the file does not exist, the punch makes it. **The punch does not truncate the file.** It writes from the start, and the end of a longer old file stays in the file.

The reader and the punch are **separate files, each with its own position**. Both are 8-bit clean: the bytes on the line are the bytes in the file. A relative path starts from the machine's folder.

The 88-HSR: a reader with a speed

A program that times its input needs a reader with a real speed. Add `?cps=N` (characters per second) or `?baud=N` (a line rate, at 10 bits for each character):

```
altairsim> CONNECT 4pio0:ja in:tape.tap?cps=300       # the 88-HSR high-speed reader
altairsim> CONNECT 4pio0:ja in:tape.tap?cps=30        # a slow reader
```

`in:tape.tap?cps=300` **is** the MITS 88-HSR. Give only one of `cps` or `baud`. Without either, the reader runs as fast as possible. The punch takes no options.

Print on a real printer

On macOS and Linux, a line can go to a print queue on your computer:

```
altairsim> CONNECT lpt0:prn printer:linewriter
```

The **Windows** builds do not have host printing yet. On Windows, send the output to a file (`out:`) or a socket (`socket:`). To see what your build can do, connect to `printer:` with no name. The program lists the queues that it can reach, or tells you that the build has no host printing.

Any line can use `printer:`, not only the 88-C700 of the Boards chapter. The bytes go into a buffer, and then to the print system as one **job**.

A printer has no "done" signal, so the program decides where a job ends. Set it on the endpoint:

Option	Ends the job	Default
`?idle=N`	after **N seconds** with nothing more printed. `0` means never	`5`
`?onff`	also on a **form feed** (the page-eject character)	off
`?max=N`	at **N bytes**, so that a program that does not stop cannot fill memory	a large number

```
altairsim> CONNECT lpt0:prn printer:linewriter?idle=15
altairsim> CONNECT lpt0:prn printer:linewriter?onff
altairsim> CONNECT lpt0:prn "printer:Generic / Text Only?idle=0&onff"
```

`?onff` with no value turns the option on. The first option that applies ends the job. **An empty buffer never prints**, so a form feed and then silence does not print a blank page. A job also prints when you

`DISCONNECT` the line, load another machine or quit.

Use a *raw* queue, which passes the bytes to the printer unchanged. A normal queue can change a printer control language that it does not recognize. Make the raw queue one time, in the printer settings of your operating system. If a queue name has spaces, put quotes around the whole endpoint, as above.

Tap a line to a log file

Add `|FILE` to any endpoint, and the program writes every byte that crosses the line, in both directions, to a text file. It writes hex and ASCII, with times. The guest cannot tell that the tap is there, and the tap never changes a byte. For this reason, you can use it on a binary transfer.

```
altairsim> CONNECT sio0:a socket:2323|bbs.hex
```

In a machine file:

```
[[board]]
id = "sio0"

  [board.unit.a]
  connect = "socket:2323|bbs.hex"
```

Connect with telnet, and use the guest. `bbs.hex` fills with the conversation:

```
# altairsim capture  socket:2323  2026-07-29 14:03:11  fmt=dump
+0.001000  TX 0000  41 54 5A 0D                                     ATZ.
+0.048213  RX 0000  0D 0A 4F 4B 0D 0A                               ..OK..
+9.100000  [DCD^]
+45.30000  [DTR_]
```

- `TX` is what the guest sent, and `RX` is what came back.
- The offset counts the bytes in a burst.
- The right column is the ASCII, with a `.` for a character that cannot be printed.
- A line in `[...]` is a change on a **modem control line**, such as carrier, DTR or RTS. `^` is a rising edge, and `_` is a falling edge.

The program empties the file each time that you connect. Add the `append` option to keep what the file holds, and to add to the end of it. `SHOW` prints the tap, and `CONFIG SAVE` saves it, so a machine file can have a line that is always traced.

The layouts and the options

The tap takes options after the file name, in the `?key=value` form:

```
altairsim> CONNECT sio0:a socket:2323|bbs.hex?fmt=cols
altairsim> CONNECT sio0:a in:reader.tap?cps=300|trace.log?fmt=jsonl
```

- **`fmt=dump`** (the default) is the layout above: one hex row on each line, in time order.
- **`fmt=cols`** puts what the guest sent on the **left**, and what it received on the **right**. A request and its reply read down the page.
- **`fmt=jsonl`** writes one JSON record for each transfer, for another program to read.
- `ts=elapsed` (the default, seconds from the first byte), `ts=wall` (the clock of your computer), or `ts=none`
- `width=N`: the bytes in each hex row. The default is 16
- `gap=MS`: how long a quiet line waits before it writes a part row. The default is 200 ms
- `pins=off`: leave out the changes on the modem control lines
- `append`: add to the end of the file. Without `append`, the program empties the file first

A log of plain text: `fmt=text`

`fmt=text` writes no hex. The file holds only the characters that the guest prints, with nothing added. Use it to keep a record of a session that you can read:

```
altairsim> CONNECT sio0:a console|session.log?fmt=text
```

In a machine file:

```
[[board]]
id = "sio0"

  [board.unit.a]
  connect = "console|session.log?fmt=text"
```

- The program writes each character to the file at the time that the guest prints it. The file is an ordinary text file, and you can read it, search it or copy it while the line is in use. For example, `tail -f session.log` in a second window follows the session. In PowerShell on Windows, the command is `Get-Content -Wait session.log`.
- The file does not hold what you type. If the guest echoes a character, the echo is in the file, as it is on a terminal.
- The file holds each byte as the guest sent it. The `[console]` settings do not change the bytes on a line (see "The console settings change bytes, and a line does not", below).

- `append` is the only option that `fmt=text` accepts. The options `ts`, `width`, `gap` and `pins` are for the hex layouts, and `fmt=text` refuses them.

A tap on a mirror

A line can have a mirror and a tap together. Put the tap last:

```
altairsim> CONNECT sio0:a console|socket:2323|session.log?fmt=text
```

The next section describes the mirror.

Mirror a line so a person can watch and take over

A **mirror** sends the line to a *socket*, in both directions. Add `|socket:PORT` to any endpoint. A second person can then type `nc localhost PORT` (or `telnet localhost PORT`; a new Mac has `nc` but no `telnet`) to see every character that the guest prints, and can also **type on the line**:

```
altairsim> CONNECT sio0:a console|socket:2323
```

The guest talks to your terminal as before. The usual use is a console that a program drives. A person connects, watches the program work, and types when they want to.

The mirror adds no echo. The watcher sees what the guest sends, and what the watcher types goes to the guest. A password that the guest does not echo stays hidden. One watcher can connect at a time.

Add `?ro` to make the mirror **watch-only**:

```
altairsim> CONNECT sio0:a console|socket:2323?ro
```

A slow watcher never slows the guest. The watcher loses some output, and the guest loses no bytes. `SHOW` prints the mirror, and `CONFIG SAVE` saves it.

`nc` and `telnet` stop when the simulator stops, and Windows installs neither. The scripts `tools/mirror-watch.sh` and `tools/mirror-watch.ps1` in the package watch a mirror, and connect again when the simulator starts again. They only watch. The chapter *The MCP server* describes them.

A mirror on a pseudo-terminal

On macOS and Linux, a mirror can go to a **pseudo-terminal** as an alternative to a socket. A terminal program, such as `screen`, `minicom` or `cu`, opens a pseudo-terminal as it opens a serial port. Add `|pty` to any endpoint:

```
altairsim> CONNECT sio0:a console|pty
sio0:a: connected to console|pty
mirror: /tmp/altairsim0 (/dev/ttys004)
```

The first mirror is `/tmp/altairsim0`, and the next one is `/tmp/altairsim1`. A second person types `screen /tmp/altairsim0` to watch the line and to type on it. The name is a link to the device in the brackets, and the program removes the link when the line disconnects.

To select the name, give a path after `pty:`. A path that is not absolute is relative to the folder of the machine file. Use this form in a machine file, because the name stays the same:

```
altairsim> CONNECT sio0:a console|pty:/tmp/console
```

In a machine file:

```
[[board]]
id = "sio0"

  [board.unit.a]
  connect = "console|pty:/tmp/console"
```

`?ro` makes this mirror watch-only also. When no program has the name open, the mirror discards what the guest prints. If `pty:` names a file that is not a link, the program refuses, and does not replace the file. Windows has no pseudo-terminal, and refuses `|pty`.

A mirror on a serial port

A mirror can go to a **serial port** on your computer. A hardware terminal, or a second computer with a terminal program, then watches the line and types on it. Add `|serial:DEVICE` to any endpoint:

```
altairsim> CONNECT sio0:a console|serial:/dev/cu.usbserial-AL009KFH
sio0:a: connected to console|serial:/dev/cu.usbserial-AL009KFH
mirror: /dev/cu.usbserial-AL009KFH at 9600 baud
```

The port is 8 data bits, no parity and 1 stop bit, with no flow control. The rate is 9600 baud. `?baud=N` selects a different rate, and the terminal must use the same one. No board programs this port, as it does

for `serial:` as an endpoint, so the rate is part of the mirror:

```
altairsim> CONNECT sio0:a console|serial:/dev/cu.usbserial-AL009KFH?baud=19200
```

In a machine file:

```
[[board]]
id = "sio0"

  [board.unit.a]
  connect = "console|serial:/dev/cu.usbserial-AL009KFH?baud=19200"
```

`?ro` makes this mirror watch-only also. Use `&` to give two options: `serial:COM3?baud=19200&ro`.

Three things are different from a socket or a pseudo-terminal:

- **The program cannot tell when a terminal is there.** The mirror sends the output to the port at all times. A terminal that you connect later shows the output that is still waiting to go, which is at most 256 KB, and the oldest output is discarded first.
- **The port sets the speed.** At 9600 baud a long listing takes time to print, and the guest does not wait for it. Only a terminal that falls behind loses the oldest text.
- **A key that you type reaches the guest only while the guest runs.** This is the same as for a socket.

If the device does not open, the program refuses, and lists the serial ports that it can see. The port cannot be the same one that the line uses.

An endpoint that `CONNECT` does not understand is an error

If the program cannot read your endpoint, it **refuses, and lists the forms that it accepts**:

```
altairsim> CONNECT sio0:a sockit:2323
no endpoint 'sockit:2323'. Try: 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 |
<endpoint>|FILE | <endpoint>|socket:PORT | <endpoint>|pty[:LINK] |
<endpoint>|serial:DEVICE[?baud=N]
```

It never uses `null` in its place. A machine that boots and prints nothing sends you to look for the fault in the wrong place.

The console settings change bytes, and a line does not

The console settings are the only part of the program that changes a byte. Every serial line is 8-bit clean. No board has a setting that clears a bit.

Setting	What it does
`upper`	changes what you type to upper case
`strip7in`	clears bit 7 of every character that you type
`strip7out`	clears bit 7 of every character that the guest prints
`crlf`	changes line endings
`echo`	shows your keys locally
`bell`	rings the bell of your terminal on `Ctrl-G`
`bsdel`	makes Backspace and Delete send the same code: `off` (the default), `bs`, or `del`. The `default` machine sets `bs`. `SHOW CONSOLE` shows the value in use
`stop`	the STOP key, a control character. The default is `Ctrl-E`. `attn` is the same setting
`base`	`hex` or `octal`: how the **monitor** prints numbers. It does not change a byte

Set them with `CONSOLE k=v`, or `SET CONSOLE k=v`. You can give several at one time:

```
altairsim> CONSOLE strip7out=on
altairsim> CONSOLE upper=on crlf=off
altairsim> CONSOLE stop=1D
```

In a machine file:

```
[console]
strip7out = true
upper     = true
crlf      = false
stop      = 0x1D
```

`[console]` is a table of its own, not a `[[board]]`. A setting that is `on` or `off` at the monitor is `true` or `false` in the file.

`stop=1D` moves the STOP key from `Ctrl-E` to `Ctrl-]`. **The STOP key must be a control character**, so that you cannot press it by accident in the middle of a word.

Why: `MEMORY SIZ?`

Boot MITS BASIC with the settings off, and it asks:

```
MEMORY SIZ?
```

MITS BASIC sets bit 7 of the last character of every message, to mark the end of the string. `E` is `45`, and with bit 7 set, it is `C5`. Your terminal prints whatever `C5` means to it.

On the real machine, the board sent all eight bits. The Teletype at the other end **did not read bit 7**, because on a Model 33 bit 7 is the parity position. It printed `E`. **`strip7out` is the terminal that does not read bit 7**, so it belongs on the console, not on the board.

A 7-bit mask on the board would fix the prompt, and it would also **damage every XMODEM transfer through that port**. XMODEM sends binary data, and bit 7 is a real bit in half of its bytes. A line can carry binary. Only the terminal knows that it shows text.

Where the console settings stop

The settings belong to the console, so they stop where the console stops:

```
altairsim> CONSOLE strip7out=on
console: strip7out=true
altairsim> CONNECT sio0:a socket:2323
sio0:a: connected to socket:2323
```

Connect with telnet, and `MEMORY SIZ?` is back. The same is true of `upper`, `crlf`, `echo`, `bell` and `bsdel`, and of a `serial:` port. The console settings are still on, but the byte no longer goes through the console. It goes from the 2SIO to the socket to your telnet client, and every step on that path is 8-bit clean.

Set the same thing on the terminal that shows the text. Every terminal program and telnet client has these settings, with its own names: strip parity or 7-bit display for `strip7out`, local echo for `echo`, and newline handling for `crlf`.

The STOP key also belongs to **your keyboard**. The program never looks for it on a socket or a serial line. A `05` that comes down a cable is part of somebody's data.

The built-in terminal has these too

The built-in `terminal` window draws the text itself, so it has the same kind of settings, under `[terminal]` in place of `[console]`:

Setting	What it does
`upper`	changes what you type to upper case
`strip7in`	clears bit 7 of every character that you type
`strip7out`	clears bit 7 of every character that the guest prints
`crlf`	adds an LF after every CR that the guest prints. Off by default
`echo`	shows your keys locally, for half-duplex software
`bell`	passes `Ctrl-G` to the terminal. On by default
`bsdel`	makes Backspace and Delete send the same code: `off` (the default), `bs`, or `del`

Set them with `SET TERMINAL k=v`, read them with `SHOW TERMINAL`, or put a `[terminal]` table in a machine file. The settings apply to the `terminal` line that is open.

```
altairsim> CONNECT sio0:a terminal
altairsim> SET TERMINAL strip7out=on
```

In a machine file:

```
[[board]]
id = "sio0"

  [board.unit.a]
  connect = "terminal"

[terminal]
strip7out = true
```

`strip7out` is useful for an **even-parity monitor**. MITS Programming System II puts parity in bit 7 of every character, so it sends a carriage return as `8D`. In the terminal window, `8D` is not a carriage return, and every line feeds without going back to the left. `strip7out=on` fixes this. `crlf=on` is for a guest that sends a CR alone and expects the terminal to add the LF.

Moving files in and out

You will want to get a file into CP/M, and to get a file back out. There is a board for this.

The Host Bridge is our own board, not a period board

MITS never made this board. No S-100 maker made it. The `hostbridge` board is **our own**, made for this simulator. The other way to move a file is a modem protocol over a simulated serial port at 300 baud. That is slow and difficult, and it is needed only because the machine is simulated.

Every other board in this program is a real product that somebody sold. This one is not, and you should know which boards are which. It is in the manual with the real boards, and it is the only one that was never sold.

It is in the **`default` machine**. Unless you wrote a machine file that leaves it out, **you already have it.**

The default port is **B0**. `BASE+0` is command and status, and `BASE+1` is data.

The three utilities, and where they are

The utilities are **on the disk**, not in the board. They are ordinary CP/M `.COM` files. They were assembled in the machine, with the machine's own assembler, from source that ships with it.

For this reason, a disk can be without them. A minidisk image that you supply is one example, and a disk that you formatted yourself is another. *Getting the utilities onto a disk that does not have them*, below, tells you what to do.

`HDIR [pattern]`	List what is on the host.
`R <hostfile> [cpmfile]`	Read host → CP/M.
`W <cpmfile> [hostfile] [B|T]`	Write CP/M → host.

`HDIR`

```
A>HDIR
    1792  07/14/26  R.COM
     640  07/14/26  HDIR.COM
   <DIR>  07/20/26  SRC/

3 files.
A>HDIR *.ASM
```

`HDIR` lists one file on each line, with its size, date and name, like `ls -l` or `DIR`. The size is a number of bytes, or `<DIR>` for a folder. A folder also keeps its `/` at the end, which shows that `R` cannot read it as a file. The list is sorted. The count at the end is the number of files in the host folder, which can be much more than a CP/M disk holds.

`HDIR` always prints the TRUE host names, with their real case and their real length. It does not print the 8.3 names that CP/M will see. That is the purpose of `HDIR`. When you cannot find the name of a file, ask `HDIR`.

A wildcard does not match a host file whose name starts with `.`, such as `.DS_Store` or `.gitignore`. This is the rule of a host shell. `HDIR` and `R *.*` skip these files. To see them, type `HDIR .*`. To copy one, type its name and a CP/M name for it: `R .gitignore GITIGNOR.TXT`.

`R`: host to CP/M

```
A>R README.TXT
A>R *.ASM
A>R SRC/FOO.ASM
A>R SRC/FOO.ASM WORK.ASM
```

Wildcards work, and they skip host files whose names start with `.`. Folders work, and **`/` and `\` both work on every host**, so you do not have to remember which computer you use. A second argument gives the name of the file on the CP/M side, instead of the automatic name.

`W`: CP/M to host

```
A>W GAME.COM
A>W GAME.COM game.com
A>W NOTES.TXT
A>W *.*
A>W *.ASM SRC
```

Wildcards work. A second argument gives the name of the file on the host side.

With a wildcard, the second argument is a host folder. `W *.ASM SRC` writes each match into the folder `SRC`, and each file keeps its CP/M name:

```
A>W *.ASM SRC
W: FOO.ASM -> SRC/FOO.ASM (text, by extension)
```

The folder must exist, because `W` does not make it. `W` checks the folder before it copies the first file. If the folder is not there, `W` prints one error and copies nothing:

```
A>W *.* NOPE
W: NOPE: no such file
```

A `/` at the end of the folder name is optional. A lone `B` or `T` is a mode, not a folder. To use a folder that has one of these names, type `B/` or `T/`.

`W` and text files: B and T

CP/M stores a file in 128-byte records and does not keep its length in bytes. A text file ends at the first `Ctrl-Z` (byte `1Ah`). CP/M does not fill the remainder of the last record. The program that wrote the file decides what is there: some programs fill it with `Ctrl-Z`, and some leave the bytes that were in their buffer. Thus `W` has two modes:

Mode	What it does	Use it for
`B` (binary)	Writes every byte of every record.	A program or data. A `.COM` file arrives unchanged, and it runs.
`T` (text)	Stops at the first `Ctrl-Z`.	Text. The file arrives at its correct length.

When you give no mode, `W` chooses from the extension of the CP/M file. These extensions are text:

`ASM` `MAC` `Z80` `HEX` `PRN` `LST` `SYM` `INC` `SUB` `TXT`

Each other extension is binary, and `.COM` is one of them. When `W` chooses text mode, it tells you:

```
A>W MON.HEX
W: MON.HEX -> MON.HEX (text, by extension)
3 records, 345 bytes
```

In a wildcard copy, `W` chooses for each file. `W *.*` brings back the source files as clean text and the programs complete, in one command.

A mode that you type is always used. Put `B` or `T` after the names:

```
A>W NOTES.DOC notes.doc T
A>W LISTING.PRN B
```

`.BAS` and `.DOC` are not in the list, because these files are not always text. MBASIC saves a program as tokens unless you tell it to save ASCII, and a word processor file can contain the byte `1Ah`. If you know that such a file is text, type `T`.

Each mode has a cost when it is used on the wrong file:

- **`B` on a text file** gives you the remainder of the last record at the end: the `Ctrl-Z`, and up to 126 bytes after it that can be `Ctrl-Z` or old data. Your editor shows them.
- **`T` on a binary file that contains the byte `1Ah` TRUNCATES it.** Byte `1Ah` is an ordinary byte in a `.COM` file (the `LDAX D` instruction). The transfer stops at the first one, and you get a file with the correct name and the wrong length.

The sandbox

The board has a **`hostdir`** property. It is the host folder that the guest can see, and it is the **only** host folder that the guest can see.

```
altairsim> SET hb0 hostdir=/tmp/xfer
```

When `hostdir` is empty, the default, it is **the folder that you started `altairsim` from.**

Where a relative `hostdir` points

`hostdir` is a path, so it follows the path rule in the machines chapter. A relative `hostdir` starts from the machine's folder, whether you type it or a machine file gives it:

```
altairsim> SET hb0 hostdir=xfer          # the xfer folder beside the machine file
```

```
[[board]]
id      = "hb0"
hostdir = "xfer"                          # the same xfer folder
```

Both mean the same folder. It moves with the machine file, so an example folder that you copy to another place still has its own transfer folder. For a built-in machine, which has no folder, a relative `hostdir` starts from the folder that you started the program from.

You do not have to work it out. `SHOW hb0` prints the value as written and the folder that it resolves to. The resolved folder is the limit:

```
altairsim> SHOW hb0
hb0  (hostbridge)

  property         value            legal
  port             0xB0             0x0..0xFE
  hostdir          xfer
  hostdir_root     /home/you/altair/disks/cpm22/xfer (read-only)
  readonly         false            true|false
```

`hostdir` is what was **written**. `hostdir_root` is where the limit **really is**. It is read-only, because it is a fact about this run, not a setting, and the program never writes it into a machine file.

`SHOW PATHS` prints the same folder, beside the base directory. If `R` cannot find a file that you are sure you put in `xfer`, read that line to see which folder the guest uses.

What the path rule means for `R` and `W`

The path rule stops at the board. **It does not reach the guest, and `R` and `W` never see it.**

At the `A>` prompt, you do not type a host path. You type a **name**, and the board finds it in `hostdir_root`, and nowhere else:

```
A>R FOO.ASM          <- hostdir_root/FOO.ASM. Never your cwd, never the machine file's
A>W RESULT.TXT       <- hostdir_root/RESULT.TXT
```

The path rule affects `R` and `W` **one time only**. It decides which folder `hostdir` names. After that, the guest has one folder, and it cannot name any other folder. The board refuses `..`, an absolute path and a drive letter, as below.

The two mechanisms look alike, but they are different:

	decides	**confines**
the path rule	where a path that you or a machine file gives points	nothing at all
`hostdir`	nothing you type	**everything the guest can reach**

This is the *only* limit in the simulator. A machine file can mount a disk from any place on your computer, but the CP/M program that runs from that disk still sees only `hostdir`.

The guest cannot leave it. The board refuses all of these, before anything touches your files:

- an absolute path
- a drive letter
- any `..` component
- a symbolic link that points outside the root

This is a **fixed limit on purpose**, not a filter that does its best. The guest runs software from 1977 that you found on the internet. You should not have to trust it.

`readonly` lets files go one way only:

```
altairsim> SET hb0 readonly=on
```

Files come **out of** the host. Nothing goes back in, and `W` fails.

The guest can write to your working folder

This is the other half of the rule, and you must know it:

By default, software in the guest can read and write the folder that you started `altairsim` from.

This is on unless you turn it off. It is a **decision**, not a mistake. `R FOO.ASM` works as soon as you unzip the package, with no setup, and the folder that you are in is usually the folder that you want. However, it is your folder, with your files in it, and a CP/M program that you did not write can reach them.

If you do not want this, point `hostdir` at a different folder, or set `readonly=on`. Each one is one command.

Names: 8.3, and how a host name becomes one

CP/M names have eight characters and an extension of three, and host names do not. `R` changes a host name in these steps:

1. It changes the host name to **upper case**.
2. It **removes the characters that CP/M cannot have in a name.**
3. It cuts the name to **8** characters.
4. It cuts the extension to **3** characters.

`R` removes these characters:

- a space
- `< > . , ; : = ? * [ ] | \ /`
- a control character, and a character that is not ASCII

All other characters stay. For example, `-`, `(` and `)` are legal in a CP/M name:

```
notes[old].txt    →   NOTESOLD.TXT
my-notes(2).txt   →   MY-NOTES.TXT
```

In the second name, `R` removes nothing. `MY-NOTES` is the first eight characters, and the `(2)` comes after them.

A second argument sets the whole name. Use it when you do not like the result of the steps:

```
A>R my-notes(2).txt NOTES.TXT
```

`HDIR` prints the **true** host names, so you can always see the name that you ask for.

Case, and why the board does not guess

The CP/M command line changes everything to upper case. The CCP does this, before your program sees the line. When `R` runs, `R readme.txt` and `R README.TXT` are **the same command**. The case is lost.

For this reason, the board does these steps, in this order:

1. **An exact match wins.** If a file has exactly the name that you asked for, the board uses that file.
2. **Otherwise, it ignores case and looks again.** `README.TXT` finds `readme.txt`. The board does this for each folder in the name too. `WORK/FOO.ASM` finds `work/foo.asm`.
3. **If more than one file matches, it REFUSES.**

Step 3 is the important one. If your folder has `readme.txt`, `README.TXT` and `ReadMe.Txt`, the board does not choose one. It does not choose the newest, and it does not choose the first. **It tells you that it cannot decide**, and it stops. A file transfer that guesses wrong and says that it worked is worse than one that fails.

Getting the utilities onto a disk that does not have them

There is a problem with all of the above: **you use `R` to get a file onto a disk, and `R` is a file on the disk.** If you boot a disk without it, such as a minidisk image that you supplied or a new disk that you made, you cannot use the board to get the program that uses the board.

You solve this with **the console**, which works before any file-transfer program does. You enter the program through the console. You do not type it. You paste it.

You cannot paste `R.COM`. It is binary, and a console carries text. The first `1Ah` in it would end the paste, and the bytes above `7Fh` would not arrive correctly. For this reason, you paste **`R.HEX`**, the file that `LOAD`

changes into `R.COM`. It is Intel HEX: colons, hex digits and line ends, about 5.2 KB of them, and every byte can be printed.

It ships with the package. The utilities are in `hostbridge/`, with the 8080 sources, the `.HEX` files and the assembled `.COM` files. The CP/M disk in `examples/cpm` already has `R.COM`, `W.COM` and `HDIR.COM`. For this reason, this section is about the other kind of disk: a new disk, or a minidisk image that you brought. For the paste below, open `hostbridge/R.HEX`.

1. **Tell PIP to write a file from the console.**

   ```
   A>PIP R.HEX=CON:
   ```

   PIP now copies what you type into `R.HEX`, until you tell it to stop.

2. **Paste all of `R.HEX` into the terminal.** Open `R.HEX` on your computer, select all of it, and paste it into the window where `altairsim` runs. PIP shows each line as it arrives, about 117 lines. That shows you that the guest receives it.

   The monitor can paste the file for you. Press `Ctrl-E` to go to the monitor, give it the file, and continue:

   ```
   altairsim> PASTE "hostbridge/R.HEX"
   pasting hostbridge/R.HEX (5231 bytes)
   altairsim> RUN
   ```

   `PASTE` sends a file of any size to the keyboard of the guest, and no character is lost. It finds a relative file name in the folder of the machine file, the same as `LOAD`. To stop a paste that is not finished, press `Ctrl-E` and type `NOPASTE`.

3. **End it with `Ctrl-Z`.**

   ```
   ^Z
   A>
   ```

   `Ctrl-Z` is CP/M's end-of-file mark on the console, and it closes the file. You are at the `A>` prompt, and `R.HEX` is on the disk.

4. **Change the HEX file into a program.**

```
A>LOAD R

FIRST ADDRESS 0100
LAST  ADDRESS 083E
BYTES READ    073F
RECORDS WRITTEN 0F
```

`LOAD` reads `R.HEX` and writes `R.COM`. It adds the `.HEX` itself, so `LOAD R` is the whole command. Check `FIRST ADDRESS 0100`. A CP/M program starts at `100h`. If that line shows a different value, the paste lost something.

5. **Use `R`, and you do not need to paste again.**

```
A>R W.COM
A>R HDIR.COM
```

`R.COM` now exists, so the other two utilities come across the board as ordinary binary files, with no HEX, no `LOAD` and no paste. `hostdir` must point at the folder that holds them. On a built-in machine, such as `minidisk`, that you started in the package folder, type:

```
altairsim> SET hb0 hostdir=hostbridge
```

For a machine file in another folder, give the path from that folder, or a full path.

Two problems of the real hardware that do not happen here

No character is lost, however fast you paste. On a real Altair, 5 KB sent to a 300-baud serial board with no flow control overruns the UART, and characters are lost in the middle of a HEX record. That cannot happen here. The simulated 6850 and 1602 UARTs deliver bytes at the line rate, but **they take a byte only when the receive register is free**. The next byte waits on the host side, and nothing is lost. A paste of any size arrives complete. This is different from the hardware on purpose, because the host connection does not lose data.

A bad paste tells you. Every Intel HEX record has a checksum, and `LOAD` checks it. A damaged line gives an error. It does not give a program that almost works.

Build them from source in the same way

When you paste `R.HEX`, you get the program. When you paste `R.ASM`, you get the program *and* the source, so that you can change it. The disk's own `ASM.COM` assembles it, and `LOAD` finishes the job as

above:

```
A>PIP R.ASM=CON:
   ...paste R.ASM, ^Z...
A>ASM R
A>LOAD R
```

It is the same method with a longer paste, about 34 KB instead of 5.2 KB. The `.COM` files in `hostbridge/` were built in this way. If you only want to *run* the utility, paste the HEX file. It is about six times shorter, and it needs no assembler on the disk.

Why one `R.COM` works on every disk

Every file operation goes through CP/M's BDOS. It does not use the BIOS or a disk parameter block. It makes no assumption about how many sectors are on a track, or where the directory is.

`R` and `W` open, read, write and close files with the same BDOS calls as any CP/M program, and the BDOS knows about your disk. For this reason, the same `R.COM`, with no changes, runs on:

- the 8″ floppy controller
- the 8 MB disk
- the minidisk
- **any BIOS that anybody writes later**, for a controller that does not exist yet

That was the goal of the design. The board is not a period board, but the software that uses it is an ordinary CP/M program, and it will work on a machine that you have not built yet.

Worked examples

These are complete sessions. **Every transcript below was captured from the program.** None of them was typed from memory.

This chapter goes through two of the machines in `examples/` in detail. Each one shows something about how the machine works, not only how to start it. Every folder in `examples/` has its own README, in Markdown and as a PDF, that says what the machine is and what to type.

More examples, with more documentation, are at https://altairsim.com.

---

1. CP/M from a floppy disk

```
$ altairsim examples/cpm/cpm22-buffered.toml
```

```
startup> RUN FF00
[console -- ^E returns to the monitor]

56K CP/M 2.2b v2.3
For Altair 8" Floppy

A>
```

What is on the disk

`DIR` lists the files:

```
A>DIR
A: L80      COM : LADDER   COM : ED       COM : ASM      COM
A: DUMP     COM : XSUB     COM : PCGET    COM : LS       COM
A: SUBMIT   COM : LOAD     COM : SURVEY   COM : VIEW     COM
A: LADDER   DAT : LUNAR    BAS : M80      COM : MAC      COM
A: MBASIC   COM : PIP      COM : STAT     COM : DDT      COM
A: MOVCPM8  COM : NSWP     COM : SYSGEN   COM : W        COM
A: ACOPY    COM : VI       DOC : STARTRK  BAS : TICTAK   BAS
A: VI       COM : CRC      COM : PCPUT    COM : AFORMAT  COM
A: STARINS  BAS : IOBYTE   TXT : R        COM : HDIR     COM
A>
```

There are 36 files: a macro assembler, a linker, Microsoft BASIC, `DDT`, `PIP`, a text editor, a disk formatter and Star Trek. It is a complete development system of 1977. `STAT *.*` gives the size of each file, and the space that is left:

```
A>STAT *.*

 Recs  Bytes  Ext Acc
   31     4k    1 R/W A:ACOPY.COM
   23     4k    1 R/W A:AFORMAT.COM
   64     8k    1 R/W A:ASM.COM
   17     4k    1 R/W A:CRC.COM
   38     6k    1 R/W A:DDT.COM
   ...
  168    22k    2 R/W A:VI.COM
  123    16k    1 R/W A:VI.DOC
    6     2k    1 R/W A:XSUB.COM
Bytes Remaining On A: 8k
```

Out, and back in

`Ctrl-E` stops the machine and gives you the monitor. Nothing is lost. `RUN` with no address continues at the instruction that the machine was about to execute.

```
A>
STOP -- the machine is still at CA9C. RUN resumes.
C0Z1M0E1I0 A=00 BC=007F DE=CA01 HL=BC0E SP=BC37 IE=1 PC=CA9C  CALL CA78
altairsim>
```

Look at the machine while it is stopped:

```
altairsim> BOARDS
altairsim> SHOW dsk0
altairsim> DUMP 100
```

To give the keyboard back to CP/M, type:

```
altairsim> RUN
```

Your `A>` prompt is where you left it.

Before you write anything

The disk is mounted **read/write**, and there is no undo. Copy the folder first. The folder has everything that the machine needs, and it boots from any place:

```
$ cp -R examples/cpm my-cpm
$ altairsim my-cpm/cpm22-buffered.toml
```

When you finish writing files, **go back to the `A>` prompt before you quit.** The BIOS keeps a track in memory, and writes it to the disk only when it next reads the console. The disks chapter tells you why.

---

2. Altair BASIC from a cassette

```
$ altairsim examples/basic4k/basic4k.toml
```

```
startup> MOUNT acr0:tape "4K BASIC Ver 3-1.tap"
acr0:tape: mounted 4K BASIC Ver 3-1.tap
startup> LOAD "LDR4K31.HEX"
loaded 20 bytes (1 page) from LDR4K31.HEX (0000-0013)
startup> RUN 0
[console -- ^E returns to the monitor]

MEMORY SIZE?
TERMINAL WIDTH?
WANT SIN? Y

742 BYTES FREE

ALTAIR BASIC VERSION 3.1
[FOUR-K VERSION]

OK
PRINT 6*7
 42

OK
```

742 bytes free. Altair BASIC was Microsoft's first product.

What the three startup lines are

They are not a boot command. **There is no boot command.** They are the three things that an operator did in 1975:

`MOUNT acr0:tape "…"`	**Put the cassette in the recorder and press PLAY.**
`LOAD "LDR4K31.HEX"`	**Toggle in the bootstrap.** Twenty bytes, entered by hand on the front-panel switches. MITS printed them in the manual.
`RUN 0`	**Set the switches to zero, press EXAMINE, then press RUN.** EXAMINE puts the switches into the program counter. Without it, RUN continues from where the machine already was.

A machine file can do what you can type at the prompt, and nothing more. For this reason, you can do the same yourself:

```
altairsim> REWIND acr0:tape
altairsim> RUN 0
```

`REWIND` is a command that **the cassette board adds**. It exists only because the machine has an ACR. You need it to load the same tape a second time, as you did in 1975.

The tape is in a startup command, not in a key

The machine file declares a front panel, a processor, a serial board, a cassette interface and some memory. **No key in it holds the tape.** A machine file's keys describe **hardware**. The cassette in the recorder is not hardware, and the board cannot control the motor. The `startup` list types the `MOUNT` for you, as you would type it. The tapes chapter tells you more.

The sense switches

The front panel's `sense` switches are set to `80`. That is `A15` up, and nothing else. The bootstrap's own printed header says why:

```
** Set A15 on (cassette load) **
** All other switches off **
```

`A15` up means *load from the cassette*. If the setting is wrong, BASIC talks to the wrong device, or to nothing. The tapes chapter tells you more.

It loaded in one second, and a real Altair took two minutes

The machine and the tape both run **as fast as possible** by default. A 300-baud cassette that took a real Altair 110 seconds loads in about one second.

To load at the real speed:

```
altairsim> SET acr0:tape rate=real
altairsim> REWIND acr0:tape
altairsim> RUN 0
```

The load now takes 110 seconds. **The guest sees the same result.** The tape has its own clock, separate from the processor's `clock_hz`, as on the real hardware. The tapes chapter describes both clocks.

Where to go next

- **The examples that this chapter does not describe:** `examples/`, and the README in each folder.
- **More machines:** https://altairsim.com has more examples and their documentation.
- **Move a file between CP/M and your computer:** the file-transfer chapter (`R`, `W`, `HDIR`).
- **Use telnet to connect to the guest, or connect it to a real serial port:** the serial chapter.
- **Look at the bus while the machine runs:** *The Debugger*.

The MCP server

```
$ altairsim <machine> --mcp
```

This command runs `altairsim` as an **MCP (Model Context Protocol) server** on stdin and stdout. An AI assistant, such as Claude or any other program that uses MCP, can then control the machine with **typed tools**. It can boot the machine, type at its console, read what it prints, set breakpoints and look at memory. It does not have to read text from a terminal screen.

The MCP server uses the same machine as the monitor. What the tools see is what `SHOW` sees.

You do not use the tools yourself. You register the server with your assistant one time, and then you tell the assistant what you want. This chapter tells you how to set it up, how to watch the assistant work, and how to run a project with it.

Set up your assistant

An MCP client needs two things: a command to run, and the fact that the server uses stdio. The command is `altairsim <machine> --mcp`. `<machine>` is a built-in name or a machine file, as on the command line.

Do not add `-x`, `-s`, `-i` or `--set` to that command. The program refuses them with `--mcp` and stops. The assistant sends monitor commands with the `monitor` tool.

Claude Code (the command line) takes it as one command. Everything after `--` is what it runs. Run it from the folder that you want the machine's files to resolve against:

```
claude mcp add altairsim -- altairsim <machine> --mcp
claude mcp list                      # confirm it registered and is reachable
```

Claude Desktop, or any client that reads a JSON configuration, needs an `mcpServers` entry with the command and its arguments. A `cwd` sets the working folder, because a desktop application has no shell to get one from:

```
{
  "mcpServers": {
    "altairsim": { "command": "altairsim", "args": ["<machine>", "--mcp"] }
  }
}
```

If `altairsim` is not on your `PATH`, give its full path as the command.

You can register more than one machine. Register each one under a different name, such as `altair-cpm` and `altair-basic`, and an assistant sees all of them. The *scope* of `claude mcp add` decides where a server is available:

- the default: only in the project folder that you added it from
- `--scope project`: in a `.mcp.json` file that moves with the folder
- `--scope user`: everywhere

The working folder is also the file-transfer folder. The host-bridge sandbox is the server's working folder. The `claude` command line uses the folder that you started from, so a relative machine path and the sandbox both point at your folder. The desktop application currently starts the server in your home folder, so there, give the machine a full path.

A first session

The `examples/ai-mcp/` folder is ready for a first session. It has a CP/M machine and a small program, `HELLO.ASM`, with one bug in it. Its README has the steps. In short:

1. Go to the folder, and register the machine from there:

```
$ cd examples/ai-mcp
$ claude mcp add altairsim -- altairsim cpm-ai.toml --mcp
```

2. Start your assistant in the same folder, and give it the job in one sentence:

> *Using the altairsim MCP tools, boot the CP/M machine and assemble and run HELLO.ASM off its > disk. It should print HELLO, WORLD. If it does not, debug it in the simulator and fix the > source.*

The assistant boots the machine, assembles and runs the program, sees the wrong output, steps through the loop to find the fault, corrects the source and assembles it again. It does all of this through the MCP tools. Say "using the altairsim MCP tools" and "off its disk", so that the assistant works on the disk in the machine, and not on a copy of the file on your computer.

Watch it work: `--mirror`

Add `--mirror socket:PORT` with `--mcp`, and you can type `nc localhost PORT` (or `telnet localhost PORT`; a new Mac has `nc` but no `telnet`) in a second window to **watch the session that the assistant**

controls, with every character that the guest prints. You can also **type on the line to take over**, and share the console:

```
$ altairsim examples/cpm/cpm22-buffered.toml --mcp --mirror socket:2323
```

This is the same mirror that the monitor has (`<endpoint>|socket:PORT`, see the chapter *Serial ports, sockets and telnet*). The assistant works as before, and the mirror changes nothing for it. What it types, and what the guest prints, also go over the socket to you. Add `?ro` to make the mirror watch-only. **Put quotes around it on the command line**, because `?` is a shell wildcard:

```
$ altairsim examples/cpm/cpm22-buffered.toml --mcp --mirror 'socket:2323?ro'
```

A watcher that connects again

`nc` and `telnet` stop each time the assistant starts the simulator again, and Windows installs neither. The package has two scripts in `tools/` that do not have these problems:

- `tools/mirror-watch.sh`, for macOS and Linux. It needs only bash.
- `tools/mirror-watch.ps1`, for Windows. It needs only PowerShell.

Run one script in a second window. It waits for the mirror, shows its output, and connects again after the simulator starts again. With no argument it watches `localhost:2323`. Give a port, or a host and a port, to watch a different mirror. `tools/README.pdf` gives the commands, and tells you how to let Windows run the script.

The scripts only watch. They do not send the keys that you type. One watcher can connect at a time. The guest prints only while it runs. Until then, the script shows `Connected` and nothing more.

A mirror for a terminal program: `--mirror pty`

On macOS and Linux, the mirror can be a **pseudo-terminal**. A pseudo-terminal looks like a serial port to a terminal program, such as `screen`, `minicom` or `cu`. A terminal program sends each key when you press it, and does not print the key a second time. There is no port number to select.

```
$ altairsim examples/cpm/cpm22-buffered.toml --mcp --mirror pty
altairsim: --mirror: open /tmp/altairsim0 (/dev/ttys004)
```

1. Read the name that the program prints. The first mirror is `/tmp/altairsim0`, and the next one is `/tmp/altairsim1`.

2. Open that name in a second window:

```
$ screen /tmp/altairsim0
```

3. To leave `screen`, press `Ctrl-A`, then `K`, then `Y`.

The name in `/tmp` is a link to the device in the brackets. The device has a different name at each start, and the link keeps the same one. The program removes the link when it stops.

To select the name, give a path: `--mirror pty:/tmp/console`. Add `?ro` to make the mirror watch-only, with quotes as for a socket: `--mirror 'pty?ro'`.

When no program has the name open, the mirror discards what the guest prints. You see the output from the time that you open it. Windows has no pseudo-terminal, and `--mirror pty` is refused there. Use `--mirror socket:PORT`.

Without the quotes, the shell tries to find a file called `socket:2323?ro`, and fails before `altairsim` sees it. (zsh reports `no matches found`.) One watcher can connect at a time.

After a `run`, the guest stops until the assistant runs it again. A character that you type while the guest is stopped waits on the line. To take over the console, ask the assistant to **`start`** the guest. After `start`, the guest runs all the time, as it does after `RUN` at the monitor, and it answers what you type at once. The assistant can still use its other tools. The guest runs until the assistant uses `stop`.

A mirror for a hardware terminal: `--mirror serial:DEVICE`

The mirror can be a **serial port** on your computer. A terminal on that port, or a second computer with a terminal program, watches the session and can type on it. This works on every platform.

```
$ altairsim examples/cpm/cpm22-buffered.toml --mcp --mirror serial:/dev/cu.usbserial-AL009KFH
altairsim: --mirror: open /dev/cu.usbserial-AL009KFH at 9600 baud
```

The port is 8 data bits, no parity and 1 stop bit, with no flow control, at 9600 baud. Add `?baud=N` for a different rate, and `?ro` to make the mirror watch-only. Quote the option, as for a socket: `--mirror 'serial:/dev/cu.usbserial-AL009KFH?baud=19200'`. On Windows the device is a name such as `COM3`.

The program cannot tell when a terminal is on the port. It sends the output at all times, at the speed of the port. The guest does not wait for it. While the guest is stopped between two `run`s, the program keeps sending what is still queued, so a long listing finishes on the terminal.

Keys that you type reach the guest only while it runs. Ask the assistant to **`start`** the guest to take over the console, as for a socket.

Keep a text log: `--log`

Add `--log FILE` with `--mcp`, and the program writes each character that the guest prints to `FILE`, at the time that the guest prints it:

```
$ altairsim examples/cpm/cpm22-buffered.toml --mcp --log session.log
```

The file holds the text that a terminal shows, with nothing added. It is an ordinary text file, and you can use it as you use any other file: show it with `cat` or `type`, open it in an editor, search it, or copy it. You can do this while the session continues.

One use is to follow the session in a second window while the guest prints:

```
$ tail -f session.log
PS> Get-Content -Wait session.log
```

The first command is for macOS and Linux, and the second is for PowerShell on Windows.

- The log holds the full session, from the first character.
- The log only records. To type at the guest, use `--mirror`. You can give `--mirror` and `--log` together.
- The program empties the file each time that `altairsim` starts.
- A file name with no folder is in the folder where you start `altairsim`.
- If the program cannot open the file, it stops with an error before the session starts.

This is the `fmt=text` tap that the monitor has (`<endpoint>|FILE?fmt=text`, see the chapter *Serial ports, sockets and telnet*).

Start a project of your own

When you want to write your own software, start in the same way as the example. **Make a folder of your own, put a machine in it, and point the assistant at that folder, not at the copy in the package.**

Put the machine and its images at the top of the folder. Copy a machine that boots the system that you want to build on into a new folder, and work there. The whole `examples/ai-mcp/` folder is a good start, with its disk and machine file. Register the server from inside that folder, as above, so that the machine file and the host-bridge sandbox both resolve there. The guest writes to the disk image as it runs, and the assistant leaves build files beside it. If you keep the copy in the package unchanged, you always have a clean one to start again from. Also keep a spare copy of the machine and its new disk, so that "start again" gives you a known state, not what the guest last wrote.

Give it local information to read. The package has the files for this: `DRIVING-WITH-AI.md`, `cheatsheet.md` and the `skills` folder. `DRIVING-WITH-AI.md` is the briefing for the assistant, with the steps for booting and debugging over MCP. The other folders in `skills` have the rules for programs that run in the machine, such as the assembler and MBASIC.

- If your assistant reads Agent Skills, install the folders in `skills` in its skills folder. The section on `DRIVING-WITH-AI.md` in the chapter *What is in the package* tells you where. The `altairsim` skill has its own copy of `DRIVING-WITH-AI.md` and `cheatsheet.md`, so do not put these files in the project folder too. Claude Code loads skills only from the places that section names. It does not load a `skills` folder at the top of the project folder.
- If your assistant does not read Agent Skills, put `DRIVING-WITH-AI.md`, `cheatsheet.md` and the `skills` folder in the project folder, and tell the assistant to read them.

Keep a second folder, for example `Reference`, for anything else that you want the assistant to use: the parts of this manual that are important to your project, and your own source material, converted to plain Markdown. An assistant reads Markdown most reliably. All of these copies go out of date when `altairsim` changes: the installed skills, the files in the project folder and the `Reference` folder. The assistant cannot tell an old copy from a current one. When you update `altairsim`, copy the skills or the files again from the new package, and update `Reference`.

Watch what it does, because it uses what it already knows. An assistant does not read the files in your folder and then do what they say. It chooses its next step from everything in front of it, and from everything that it learned in training. What it learned is very large, and one line in a local file has little weight against it. For this reason, when the job looks like a common one, such as booting CP/M, assembling a file or copying something to a disk, the assistant first tries the way that it has seen most often, even when the files in your folder say something else. For example, when you ask it to get a program onto a disk, it tries a host tool such as `cpmtools`. That tool does not understand the hard-sectored disks of the Altair. The way that works is to build inside the machine over the host bridge, as `skills/altairsim-cpm-build/SKILL.md` says. The longer a session runs, the more the assistant goes back to the common way.

For this reason:

- Watch what it sends and reads back, with `--mirror`. Do not trust its own summary of what it did.
- Correct it at the first wrong step, before it builds on that step.
- Point it at the file that you mean, and tell it to use that file and only that file.
- Be most careful when its answer looks the most usual. That is when a habit has most probably taken the place of something that is special about your machine.

Start and stop each session in the same way. A short instruction that you give every session saves time. At the end, tell it to clean up: close any processes that still run, write down what it learned, and save where you are in a note in the folder. At the start, tell it to read that note and the `Reference` folder, boot

the machine, and tell you where you stopped. The note in the folder lets a session continue. Without it, the assistant must work out the state again each time, and it gets it wrong.

Do not rename the project folder in the middle of a project. Claude Code keeps its memory of the project under the full path of the folder, outside the folder itself. After a rename, the folder looks like a new project to it. The registration from the section above is also tied to that path by default. `claude mcp list` in the renamed folder no longer shows `altairsim`, and the assistant cannot reach the machine until you run `claude mcp add` again. `--scope project` prevents this, because the registration is then in a `.mcp.json` file that moves with the folder.

If it happens, register the server again, tell the assistant the old and the new folder names, and tell it to start from the note in the folder.

What the assistant can do

The tools let an assistant operate the whole machine:

- List the available board types, and every property of each type, with its type, its default and its legal range.
- List the boards in the machine, get and set any property, and add a board.
- Mount a disk or a tape, and connect a serial line to an endpoint.
- Examine, deposit, fill, search, save and disassemble memory.
- Run the machine, type at its console and read what it prints.
- Start the machine and let it run between tool calls, for example to run a server on the guest, or two machines that talk to each other. Then stop it.
- Step it, set breakpoints, and read the bus history and the interrupt lines.
- Save the state of the whole machine in a snapshot, and restore it.

When no tool does what it needs, such as a conditional breakpoint or an octal listing, the `monitor` tool runs any monitor command and returns its text.

This manual has no tool reference, because the server describes its own tools. `tools/list` returns every tool, and `board_types` returns every board type. Their answers always match your copy of the program. A new board is available to an assistant as soon as it is in the program, because the tools come from the same description of the board as its machine-file keys.

If you write your own client

Most people use an assistant, and the assistant reads the tools itself. If you write a program that talks to the server directly, read `DRIVING-WITH-AI.md`. It describes the protocol, the `run` tool and its stop reasons, how to stop a `run` early, and the `status` tool. These points are the most important:

- The server uses line-delimited **JSON-RPC 2.0**. Send `initialize`, and then `tools/call`.
- A tool result has text in `content`. A tool that reports values also gives them as JSON fields in `structuredContent`, such as `running` and `stop_reason` from `status`. A program reads `structuredContent` and does not parse the text.
- A machine named on the command line is loaded, but **its `startup` list does not run**. The client boots it with `run`.
- One serial unit holds the console. The tools type on its line and read its output. When the `connect` tool or `CONNECT` connects another unit to `console`, that unit takes the console. The unit that held it goes to `null`, and the reply says `console taken from` that unit, as in the monitor. To take the console back after you connect its unit to another endpoint, such as `pty`, connect the unit to `console` again. The server writes only JSON-RPC to stdout.
- A `run` **never blocks**. It stops when `until` matches, when the guest waits at a prompt, or at `timeout_ms`. `timeout_ms` is a limit, not a wait.
- `stop_at` stops a `run` when the PC gets to an address. The instruction at that address does not run. The reply gives `stopped` as `stop-at`. The stop is for that one `run` only. It is not a breakpoint, and `BREAK` does not list it. If the PC is already at the address, the guest runs until it gets back there.
- `start` returns at once and **leaves the guest running** until `stop`, a `HLT` or a breakpoint. All other tools work while it runs. `status` tells you if it still runs, and if not, why it stopped.
- JSON has no hex form, so `from` and `stop_at` are decimal numbers: `65280` is `FF00`. If you send a string such as `"0xFF00"`, the server refuses the call and tells you the number to send.
- A control byte is a JSON `\uXXXX` escape: `\u0003` is `Ctrl-C`. `\x03` is not JSON.
- Some clients send the text as you write it, and do not make a JSON escape from it. With such a client, use the `TYPE` command through the `monitor` tool. `TYPE` has its own escapes: `\r`, `\"`, `\^C` for `Ctrl-C` and `\xHH` for a byte. `HELP TYPE` gives the full list.

Troubleshooting

Most problems here are not faults. The machine does what a real machine did, in a way that you did not expect. This chapter explains the most common cases.

`Ctrl-C` does not stop the machine

It is not supposed to.

`Ctrl-C` belongs to the guest. CP/M reads it to do a warm boot, and BASIC reads it to stop a program. If the stop key were a key that the guest also uses, the guest would take it, and you could not stop the machine.

`Ctrl-E` is the stop key. It presses the STOP switch on the front panel. The program reads it before the guest gets the byte, and **no program in the machine can turn it off, ignore it or take it from you.**

If a guest needs `Ctrl-E` for something of its own, move the STOP key:

```
altairsim> CONSOLE stop=1D
```

The STOP key is then `Ctrl-]`. It must be a control character.

I pressed STOP, and now the machine does not move

It is **stopped**, not broken. That is what STOP does, and nothing is lost.

STOP stops the processor and gives you the monitor. It is not RESET, and it is not POWER. The registers, the memory and the disk are as the processor left them. The monitor printed where the machine stopped, in the line "*still at* ...".

```
altairsim> RUN
```

`RUN` with no address continues at that instruction, and your `A>` is where you left it. `RUN <addr>` is different. It loads the PC first, so it *starts again* at that address.

While the machine is stopped, you can look at it. `REGS`, `EXAMINE`, `DUMP`, `DISASM` and `STEP` all work at this prompt, on a machine that does not move. See *The Debugger*. Use `BREAK` to stop at a place that you cannot reach by hand.

My file was not written, or the disk lost my changes

You quit too early.

The CP/M in this package has a **track-buffering BIOS**. It keeps a whole track in memory, and writes it to the image only when it next reads the console. The simulator does not cause this. The BIOS works this way, and that is why it was fast. Not every CP/M does this. Some write each sector at once. You cannot tell from the prompt which kind you have, so think of it as a buffered one.

Go back to the `A>` prompt before you quit or copy the image. When CP/M waits for a key, the buffer is on the disk. If you stop the program right after a file operation seems to finish, the last track is not on the disk. The disks chapter gives the details.

The guest reports a disk error

The simulator adds no disk errors. A simulated drive has no dust, no worn surface and no head out of alignment. When the guest reports `Bad Sector`, `DISK WRITE ERROR` or a directory that makes no sense, the cause is the setup of the drive or the image itself. Check these things, in this order:

1. **A `media` line on the drive** in your machine file. Remove it, unless the drive holds a blank disk. See below.
2. **Write protection.** Type `SHOW MOUNTS`. A write to a disk that shows `(write-protected)` fails in the guest, and CP/M reports it as `Bad Sector`. The disks chapter tells you why.
3. **The image.** An image can be damaged, or it can be for a different machine. A download can be cut short, and an archived disk can have sectors that could not be read when somebody made the image. Try the image in the machine that it came from, or get another copy.

`media` is the first thing to check

The program finds the format of a disk from the size of the file: the number of tracks, the sectors on each track and the bytes in each sector. `media` turns this off, and sets the format by name. If the name is wrong, the drive reads and writes the file at the wrong places:

- **A format that is too small** gives the drive fewer tracks than the file has. The first tracks work, so the directory and the first files are correct. The first read or write past the last track fails. CP/M uses the layout in its own BIOS, so `STAT` and `LS` still show the full disk and a lot of free space. The error can come late, after many good copies.
- **A format with a different layout**, such as a different sector size or density, puts every sector at the wrong place in the file. The errors start at the first read. The guest reads wrong data, sees a directory that makes no sense, or reports `Bad Sector`.

For example, this drive holds an 8 MB image, and `media` makes it an 8″ floppy with 77 tracks:

```
[[board.drive]]
unit  = 1
media = "8in"            # wrong: this file is an 8 MB disk
mount = "b-8mb.dsk"
```

CP/M copies the first files to B: with no error. Later, a copy that writes past track 76 fails with `DISK WRITE ERROR`, and it leaves `.$$$` files. Remove the `media` line, and the program finds the 8 MB format from the size of the file.

Use `media` only when the size of the file cannot decide the format:

- a blank disk from `create = true`, because an empty file has no size
- an image that is cut short, or that has a layout that the program does not know

For every other disk, leave `media` out. After you fix the machine file, delete the `.$$$` files and any file that you are not sure of, and copy again. A write that failed can leave a file that is only partly written.

The disk image changed, and I did not want it to

There is no undo. **The image is mounted read/write, as on a real machine.** A CP/M that cannot write its disk cannot save your work.

There are two answers. Use one *before* you experiment:

```
altairsim> MOUNT dsk0:drive0 file.dsk WP
```

`WP` refuses every write at the controller, so your file is safe whatever the guest does. Use it for a disk that you only want to **read**. The guest is never told that the disk is write-protected. The controller has no bit for it, and CP/M has no error for it. For this reason, a program that tries to write to a `WP` disk does not fail in a controlled way. The disks chapter tells you why.

The other answer is to copy the folder. It has a machine file and an image in it, and it boots from any place:

```
$ cp -R examples/cpm my-cpm
```

`MOUNT` says "no such file", and the file is there

Look at *which machine you run*.

A relative path starts from the machine's folder, which is the folder that the machine file was loaded from. It does not start from the folder that you started the program from. For example, you go to a folder that has a disk, start a machine from another folder, and type `MOUNT dsk0:drive1 that.dsk`. The program looks for the disk beside the *machine* file, not in your folder. `SHOW PATHS` prints the folder that it uses.

The program works this way on purpose, so that an example folder boots from any place. The machine file says `cpm22.dsk`, and that means *the image beside me*. You can move the folder, or run it from three levels up, and it still finds its disk. A name that you type finds the same folder, so the disk that came with the machine and a disk that you mount by hand are in one place.

Every prompt ends in a wrong character

```
MEMORY SIZ?
```

MITS BASIC sets bit 7 of the last character of every message, to mark the end of the string, and your terminal prints the result. The `E` is there. It arrived as `C5`, not `45`.

```
altairsim> CONSOLE strip7out=on
```

The real Teletype ignored bit 7 and printed the `E`. `strip7out` makes your terminal do the same. The serial chapter tells you why the fix belongs on the console, and **never** on the board.

Nothing appears when I type, or the guest does not see my keys

Find out which unit has the keyboard:

```
altairsim> SHOW CONSOLE
```

Only one unit can hold the console. If the console is on a unit that the guest does not read, you type into a board that nothing reads. Connect the correct unit. When you connect a second unit to `console`, it *takes* the console from the first unit, and says so.

The machine runs very fast, and a cassette loads in one second

The processor and the tape both run **as fast as possible** by default. They have separate settings:

```
altairsim> SET cpu0 clock_hz=2000000
altairsim> SET acr0:tape rate=real
```

`clock_hz = 2000000` gives the processor the speed of a real 2 MHz Altair. `rate = real` makes a tape load at its real speed, so a cassette takes its real 110 seconds. Each setting is independent of the other. The tapes chapter describes both. The fast default and the real speed are both correct.

A file transfer times out: `PCGET`, `PCPUT`, XMODEM

Slow the processor down. A transfer with the outside world needs the real crystal.

```
altairsim> SET cpu0 clock_hz=2000000
```

This section tells you why, because the same cause explains many problems.

A guest program has no clock. It counts instructions. `PCGET` measures a second in the same way as every program of the period: it runs a loop and counts the turns. Its own source says so:

```
MSEC    lxi  d,(159 shl 8)   ;49 cycle loop, 6.272ms/wrap * 159 = 1 second
```

That count is one second *only* with a 2 MHz crystal. `PCGET` waits three seconds for a block header. At full speed, your computer runs those three seconds of **T-states** in a few tens of **milliseconds**. The sending program at the other end of the line still counts real seconds. `PCGET` decides that the sender is dead, sends a NAK, clears the line, and stops. Nothing is broken. The two ends no longer use the same clock.

This is the exact limit. Timing inside the machine is correct at any speed, because everything in the machine counts the same T-states. A transfer to your computer has **one end inside the machine, and one end outside it**. Only the real crystal makes the two ends agree.

For this reason, use full speed for everything that the machine does by itself, and the real 2 MHz for everything that it does with you. You can set it back afterward. The built-in file-transfer board does not have this problem, because it has no timeouts.

Two boards use the same port

```
altairsim> SHOW BUS CONTENTION
```

This command names the boards. To see which board decodes each port, type:

```
altairsim> SHOW BUS IO
```

On a real S-100 machine, this was two boards set to the same address, and a bus that gave wrong values. Here, the program lists the conflicts.

An `IN` from a port returns `FF`, and I expected a value

No board decodes that port. When no board drives the bus, the bus floats high, and a floating bus reads `FF`. This is not an error, and the machine does not tell you. A real machine did not either.

To find which board *would* answer, without a bus cycle, type:

```
altairsim> WHO IO 10
```

A **hang** can have the same cause. A machine that you built yourself starts, prints nothing, and never returns. It reads a port for a board that is not there, and gets `FF` forever. Tell the bus to report it:

```
altairsim> SET BUS UNCLAIMED=WARN
bus: unclaimed=WARN
altairsim> RUN
warning: IN 10 -> FF at PC=0043: no board decodes port 0x10. it floated to 0xFF.
```

`WARN` prints the port and the address of the instruction, and the machine continues. `HALT` stops the machine at that instruction, so that `REGS` and `DUMP` show it as it stopped. The setting watches I/O only, because a program often reads memory where nothing is. It reports each missing port once in each run. The default is `SILENT`. Turn it on when you need it.

`RESET` did not clear memory

It is not supposed to.

`RESET` is the RESET* line of the bus. It does what that line did: the processor goes to address zero, and the boards go back to their power-on state. RAM keeps its contents. The real machine did not clear it, and

the program does not clear it.

Only `POWER` loses memory. That is the difference between pressing a button and pulling the plug, and the program keeps that difference.

macOS refuses to run the program

macOS checks the program with Apple the first time that you start a new copy. If the Mac is offline, the check fails. macOS then shows one of these messages:

- Newer versions of macOS: a dialog with the title "*altairsim*" *Not Opened* and the text *Apple could not verify "altairsim" is free of malware*.
- Older versions: "*altairsim*" *cannot be opened because the developer cannot be verified.*

Press Done. Do not press Move to Trash. Move to Trash is the default button, so the Return key deletes the program.

Connect the Mac to the internet, and start the program again. If you cannot connect it, remove the mark that macOS put on the download:

```
$ xattr -dr com.apple.quarantine ./altairsim
```

You can also allow the program in System Settings. This way needs an administrator password.

1. Press **Done** in the first dialog.
2. In System Settings, open Privacy & Security.
3. Press **Allow Anyway**.
4. Start the program again. A second dialog asks if you want to open it.
5. Press **Open Anyway**. Do not press Move to Trash. It is the default button in this dialog also.

Glossary

ACIA: Asynchronous Communications Interface Adapter. The Motorola 6850 chip on the 88-2SIO serial board. A program can see two of its registers: a status register and a data register. A program does everything with a serial port on this machine through one of those two registers.

ACR: Audio Cassette Recorder interface. The MITS 88-ACR, a board that wrote bits to an ordinary audio cassette and read them back. People who could not pay for a floppy disk loaded BASIC with it, and in 1976 that was most people.

ATTN: See **STOP**. ATTN is the old name of the STOP key. `CONSOLE attn=` still works, and it means the same as `CONSOLE stop=`.

backplane: The circuit board with the connectors that every board plugs into. It is not one of the boards. A machine file adds no backplane, because a machine file describes what is in the backplane. On an Altair, the backplane is the S-100 bus: eighteen slots of one hundred pins each, all connected in parallel. There is no chipset. The backplane *is* the machine.

bank switching: A way to give the processor more memory than it can address. Several banks of memory use the same addresses, and a switch selects one bank at a time. An 8080 can address 64K and no more, so a machine with 128K of RAM shows the processor one part of it at a time.

BDOS: Basic Disk Operating System. The middle layer of CP/M, which handles files, directories and the console. A CP/M program asks the BDOS for what it needs, and does not know which hardware answers.

BIOS: Basic Input/Output System. The bottom layer of CP/M, and the only part that knows what machine it is on. To move CP/M to a new computer, you rewrite the BIOS. The BDOS and the CCP do not change. A track buffer is also in the BIOS, *if the author put one there*. For this reason, on some versions of CP/M and not on others, a file can be "written" and not yet be on the disk. See the disks chapter.

board: Anything that plugs into the backplane: memory, a serial port, a disk controller, the front panel, or the processor itself. This program uses the word everywhere. `BOARDS` lists them, and `[[board]]` adds one in a machine file. A board is the thing that you add, remove, `SHOW` and `SET`. See also **card**, which is the same object.

card: The same object as a **board**. The period hardware and its manuals said "card". This manual uses the word only when the sentence is about the physical object that somebody bought, put chips into and set jumpers on. Everywhere else, it says board.

CCP: Console Command Processor. The top layer of CP/M, which prints `A>` and runs what you type. A large program can write over it, and CP/M loads it again from the disk afterward. That load is the warm boot.

contention: Two boards that answer the same address. On a real S-100 machine, both would drive the data bus at the same time, and you would get a byte that is neither board's value, sometimes and not always. That is very hard to find. `SHOW BUS CONTENTION` names the boards.

CP/M: Control Program for Microcomputers. The disk operating system of Digital Research, and the reason that the 8080 was important. A program written for CP/M ran on 8080 machines from many different makers.

CUTS: Computer Users Tape System. The cassette format of the Sol-20, and a faster version of Kansas City. At its default of 1200 baud, it uses tones one octave lower, **1200/600 Hz**, and puts one bit in one cycle (a "1") or in a half cycle (a "0"). Its slower 300-baud mode is Kansas City, 2400/1200 Hz. The Sol's UART does both, and the guest selects one with bit D5 of `OUT 0FAh`.

DBL: Disk Boot Loader. The boot PROM of the MITS floppy disk controller, at `FF00`. It reads the program at the start of track 0 into memory, and jumps to it. On a CP/M disk, that program loads CP/M. **There is no `BOOT` command on an Altair.** You set the address switches to `FF00`, press EXAMINE to load them into the program counter, and then press RUN. DBL is the program that then runs.

decode: What a board does when it recognizes an address as its own, and answers. A board that does not decode an address stays silent, and another board can answer. Which board decodes which address is the main question of how an S-100 machine is put together.

DMA: Direct Memory Access. A board takes the bus from the processor and reads or writes memory itself, without the processor. It is faster, and it can cause faults that are difficult to find.

endpoint: In this program, the thing at the far end of a serial unit's cable, for example `console`, `null`, a TCP socket or a real serial port. The serial chapter has the complete list.

floating bus: What the data bus reads when nothing drives it. On the S-100 bus it floats high, so an `IN` from a port that no board decodes returns `FF`. This is not an error. It means that no board is there.

front panel: The switches and lamps on the front of the Altair: the address switches, the data lamps, and DEPOSIT, EXAMINE, RUN, STOP and RESET. Before there was a terminal, the front panel *was* the user interface, and you toggled in your bootstrap through it, one byte at a time. In this program, it is a board like any other.

FSK: Frequency Shift Keying. A way to encode bits as two audible tones, one for a zero and one for a one. The ACR used it to put data on a cassette, and it is the reason for the sound of a tape that loads.

The tones were **not** the same on all machines, and this is important. The 88-ACR uses 2400/1850 Hz, and holds a tone for the whole bit. Kansas City counts *whole cycles* of 2400/1200 Hz for each bit. CUTS, at its default of 1200 baud, does the same one octave lower (1200/600 Hz, with a half cycle for a "0"). A board can read only the modulation that its own modem was built for. For this reason, the program refuses a recording in the wrong modulation, and does not decode it. See the tapes chapter.

guest: The software that runs on the simulated machine, such as CP/M, BASIC or a program that you wrote. The guest cannot tell that the machine is simulated.

hard-sector: A floppy disk where physical holes in the disk mark the sector boundaries, and the controller counts them as they go past. The MITS floppy disks are hard-sectored. A soft-sectored disk has one hole, and finds its sectors from marks written in the data. Soft sectors became the common method.

IntAck: Interrupt Acknowledge. The bus cycle that the 8080 runs when it accepts an interrupt. The board that interrupts puts one instruction on the data bus during that cycle, and the processor executes it. It is almost always an `RST`.

Kansas City standard: The 1975 agreement on how microcomputers should record data on ordinary audio cassettes, named after the meeting where it was agreed. A one is eight cycles of 2400 Hz, and a zero is four cycles of 1200 Hz. Both take the same time. For this reason, a receiver counts cycles and does not depend on a clock, so a tape that plays 5% slow still reads. It runs at 300 baud. See also **CUTS**, its faster version, and **FSK**.

machine file: The `.toml` file that describes a machine: its boards, their settings, and the commands to run when it starts. The configuring chapter describes the format.

MCP: Model Context Protocol. The protocol that an AI assistant uses to call structured tools. `altairsim --mcp` uses it, so that an assistant can control the machine directly. See the MCP chapter.

monitor: The `altairsim>` prompt. It is not a program that runs in the machine. It is you, in front of the machine, with more controls than the real front panel had.

PHANTOM*: An S-100 line that tells memory boards to stop answering. When a board pulls it, a ROM can answer at an address that a RAM board also decodes, with no contention. In this way, a boot PROM can cover RAM and then get out of the way. The asterisk means that the line is active low, as most lines on this bus are.

pINT: The S-100 interrupt request line. Any board can pull it. The processor sees it, if interrupts are enabled, and runs an IntAck cycle to find out which board interrupted and why.

PROM: Programmable Read-Only Memory. A chip with a program in it that stays when the power is off. The Altair's boot loader is in one. Without it, a disk machine could not start, because something must already be in memory to read the disk.

RST: Restart. A one-byte 8080 instruction that calls a fixed low address: `RST 0` to `RST 7`, at `0000`, `0008` and so on to `0038`. It is one byte, so it fits in an IntAck cycle. That is what it is for.

S-100: The bus. One hundred pins and eighteen slots. MITS designed it for the Altair, and then other makers used it. It was an early open standard in personal computing, and for this reason, a board from one company worked in the machine of another company.

sector: The smallest part of a disk that you can read or write. On these floppy disks, a sector is 137 bytes, and 128 of them are yours.

SENSE switches: The top eight address switches (A8 to A15) on the front panel. A program can read them as an input port. Software used them for settings before there was any other place for them: which port is the console, how much memory to use, or whether to load from tape.

STOP: The STOP switch on the front panel, and the key that presses it, `Ctrl-E` by default. It stops the processor and gives you the monitor. It does not change the machine. It is not RESET, and it is not POWER. For this reason, `RUN` with no address continues at the instruction that the machine was about to execute. The program reads the key before the guest sees the byte, so no guest program can take it from you. Move it with `CONSOLE stop=`. The older `attn=` still works.

T-state: One clock cycle. Every 8080 instruction takes a known number of them, and this program uses them to measure time. At 2 MHz, a T-state is 500 nanoseconds.

TDRE: Transmit Data Register Empty. The bit in the 6850's status register that means "the board has room for another character". A program that prints reads this bit until it is set. A serial port connected to `null` has it set all the time. For this reason, a program that writes to nothing works.

track: One ring of sectors on a disk. The head steps in and out to reach a track, and does not move again to reach the sectors on it. For this reason, a BIOS with a buffer keeps a whole track at a time: after one seek, it can read or write every sector of the track.

UART: Universal Asynchronous Receiver/Transmitter. The chip that changes a byte into a series of bits on a wire, and back again. The ACIA is one.

unit: One channel of a board that moves characters, like the connector on the back of the board. A 2SIO has two units, `a` and `b`, and you connect each one separately. `CONNECT sio0:b` names the board and then the unit.

VI0–VI7: The eight vectored interrupt lines of the S-100 bus, served by the 88-VI/RTC board. **VI0 has the highest priority.** A board that pulls VI*n* gets `RST` *n*, and the interrupt board decides which board wins when two pull at the same time.

warm boot: CP/M loads its own top layer (the CCP) again from the disk, because the program that had finished could write over it. `Ctrl-C` at the `A>` prompt does a warm boot on purpose. It is not a reset, and the machine does not stop.

Boards and their parameters

Every key below is a key you may write in a machine file, and the *same* key you may `SET` at the monitor prompt. That is not a coincidence and it is not a convention: a board's properties **are** its TOML schema, so there is nothing here that could disagree with the program.

Numbers follow the one rule: **on the wire → hex, never on the wire → decimal.** A port is hex; a baud rate and a drive count are decimal. The defaults below are printed in each property's own base.

The catalogue is **grouped by function** — CPU, memory, disk, serial, and so on — and within a group the boards are in **alphabetical order**.

CPU

Type	What it is
`8080`	MITS 88-CPU: 8080A CPU board
`8085`	Generic 8085 CPU board
`z80`	Generic Z80 CPU board

Memory

Type	What it is
`bankmem`	Bank-switched RAM (Vector, Cromemco, North Star, ExpandoRAM)
`memory`	RAM/ROM board: plain, unbanked memory regions
`v2z80rom`	S100Computers V2 Z80: paged monitor EEPROM

Disk

Type	What it is
`16fdc`	Cromemco 16FDC: floppy controller + console UART, RDOS 2.52
`64fdc`	Cromemco 64FDC: floppy controller + console UART, RDOS 3.12
`dcdd`	MITS 88-DCDD: 8" hard-sector floppy controller
`disk1`	CompuPro Disk 1: uPD765A floppy controller with DMA
`disk1a`	CompuPro Disk 1A: uPD765A floppy controller with DMA
`dualide`	S100Computers IDE-AB: two CompactFlash sockets for CP/M 3
`dualsd`	S100Computers Dual SD: two microSD sockets for CP/M 3
`fdcplus`	FarmTek FDC+: serial drive (drive types 6, 7)
`hdsk`	MITS 88-HDSK Datakeeper: Pertec hard disk controller
`icom`	iCOM FD3712/FD3812: 8" floppy controller with boot PROM
`mds`	MITS 88-MDS: 5.25" minidisk controller
`mdsa`	North Star MDS-A: single-density 5.25" floppy controller
`mdsad`	North Star MDS-A-D: double-density 5.25" floppy controller
`tarbell`	Tarbell #1011: single-density floppy controller
`tarbelldd`	Tarbell #2022: double-density floppy controller
`versafloppy`	SD Systems VersaFloppy I/II: WD177x floppy controller

Serial

Type	What it is
`2sio`	MITS 88-2SIO: two 6850 serial ports
`gsio`	Generic SIO: two strap-configurable serial channels
`io4`	SSM IO-4: two serial + two parallel ports
`pmmi`	PMMI MM-103: Bell 103 modem
`propio`	S100Computers Console IO: Propeller console serial port
`sbc`	SD Systems SBC-100/200: Z80 single-board computer
`sio`	MITS 88-SIO: one serial port (COM2502 UART)
`turnkey`	MITS 8800b Turnkey Module: boot PROM, serial, auto-start

Tape

Type	What it is
`acr`	MITS 88-ACR: audio cassette interface
`uio`	MITS 88-UIO: serial port + cassette interface

Parallel and printer

Type	What it is
`4pio`	MITS 88-4PIO: up to four 6820 parallel ports
`c700`	MITS 88-C700: Centronics line-printer controller
`d7a`	Cromemco D+7A: analog + parallel I/O, joysticks
`lpc`	MITS 88-LPC: 88-LP line-printer controller
`music6`	Newtech Model 6 Music Board: 6-bit D/A and speaker
`pio`	MITS 88-PIO: 8-bit parallel port

Video

Type	What it is
`cadzilla`	CADzilla: HD63484 ACRTC graphics board with a Bt453 RAMDAC
`dazzler`	Cromemco Dazzler: color graphics
`dazzler2`	Cromemco Dazzler II: color graphics, on-board RAM
`vdb8024`	SD Systems VDB-8024: 80x24 video terminal board
`vdm1`	Processor Technology VDM-1: 16x64 memory-mapped video

Systems

Type	What it is
`sol`	Processor Technology Sol-PC I/O: serial, keyboard, tape

PROM programmer

Type	What it is
`pb1`	SSM PB1: 2708/2716 EPROM programmer

Other

Type	What it is
`fp`	Altair front panel: the SENSE switches at IN 0FFH
`hostbridge`	Host Bridge: file transfer to the host (not a period board)
`rtc100`	SciTronics RTC-100: battery-backed clock/calendar
`ss1`	CompuPro System Support 1: interrupts, timer, clock, serial
`virtc`	MITS 88-VI/RTC: vectored interrupts + real-time clock

CPU

`8080`

MITS 88-CPU: an 8080A CPU board. Decodes nothing -- it drives the bus

Units: `8080` (cpu)

Board properties

Key	Kind	Default	Legal	Meaning
`clock_hz`	int	`0`	`0` .. `100000000`	Crystal on the board. 0 runs flat out -- as fast as the host can.
`idle`	bool	`true`	`on` | `off`	Stand down when the guest is only polling an empty keyboard. On by default -- the guest cannot tell, and a prompt stops burning a core.
`achieved_hz`	int	—	—	LIVE: T-states per real second the run loop last reached -- the crystal you got, beside the one you asked for. Read-only; 0 until it has run. **(read-only — not a key you may set)**

`8085`

Generic 8085 CPU board. Decodes nothing -- it drives the bus. The 88-CPU's twin, with an 8085 core (RIM/SIM + TRAP/RST 5.5/6.5/7.5)

Units: `8085` (cpu)

Board properties

Key	Kind	Default	Legal	Meaning
`clock_hz`	int	`0`	`0` .. `100000000`	Crystal on the board. 0 runs flat out -- as fast as the host can.
`idle`	bool	`true`	`on` | `off`	Stand down when the guest is only polling an empty keyboard. On by default -- the guest cannot tell, and a prompt stops burning a core.
`achieved_hz`	int	—	—	LIVE: T-states per real second the run loop last reached -- the crystal you got, beside the one you asked for. Read-only; 0 until it has run. **(read-only — not a key you may set)**

`z80`

Generic Z80 CPU board. Decodes nothing -- it drives the bus. The 88-CPU's twin, with a Z80 core

Units: `z80` (cpu)

Board properties

Key	Kind	Default	Legal	Meaning
`clock_hz`	int	`0`	`0` .. `100000000`	Crystal on the board. 0 runs flat out -- as fast as the host can.
`idle`	bool	`true`	`on` | `off`	Stand down when the guest is only polling an empty keyboard. On by default -- the guest cannot tell, and a prompt stops burning a core.
`achieved_hz`	int	—	—	LIVE: T-states per real second the run loop last reached -- the crystal you got, beside the one you asked for. Read-only; 0 until it has run. **(read-only — not a key you may set)**

Memory

`bankmem`

S-100 bank-switched RAM. One card, four decoders (card=vector|cromemco64kz|northstar|expandoram2): a write-only select port swaps which RAM plane(s) drive the bus. Each card owns its own decode -- one-hot select (Vector 40), 8-bit bank mask (Cromemco 40), on/off+one-hot toggle (North Star C0), or PROM page-select (ExpandoRAM II FF, approximated)

Board properties

Key	Kind	Default	Legal	Meaning
`card`	enum	`vector`	`vector` | `cromemco64kz` | `northstar` | `expandoram2`	which banked card this is -- each owns its own decode: vector | cromemco64kz | northstar | expandoram2
`port`	int	`0x40`	`0x0` .. `0xFF`	the write-only bank-select port. Card default (vector/cromemco 40, northstar C0, expandoram2 FF); overridable, as the real boards relocate it
`banks`	int	`8`	`1` .. `10`	how many switchable banks/planes/pages this subsystem carries (one per board). Card-capped: vector/cromemco 8, northstar 6, expandoram2 10. The common region (partition) is not a bank. Setting `ram` derives this
`partition`	enum	`none`	`none` | `ex48` | `ex32`	ExpandoRAM II common-memory partition (the EX-48/EX-32 PROM): none | ex48 | ex32. ex48 = 48K banked (0000-BFFF) + 16K common (C000-FFFF); ex32 = 32K banked + 32K common (8000-FFFF). The common region is identical in every bank -- what a banked CP/M's resident OS and bank-switch routine live in. `none` (default) is a whole 64K plane
`ram`	int	`512`	`1` .. `640`	total board RAM in KB. Sets the bank count from the current partition: with ex48, 256 -> 5 banks; with none (whole plane), 256 -> 4 banks of 64K. The real ExpandoRAM II is 64K (16K chips) or 256K (64K chips)
`honors_phantom`	enum	`none`	`none` | `read` | `all`	A JUMPER. Another board pulls PHANTOM* (a boot PROM shadowing the RAM this card sits under) -- do I switch off? none | read | all. `read` keeps answering writes so a cold-boot loader falls through to this RAM while the PROM shadows reads (an ExpandoRAM II under the SBC-200 boot PROM). Default none -- a plain banked-RAM machine has no PROM overlay
`active`	string	—	—	the live bank(s) right now -- the guest sets this by writing the select port. Read-only here **(read-only — not a key you may set)**
`fill`	enum	`random`	`zero` | `random`	RAM contents at power-on: zero | random (real RAM is not zeroed)
`seed`	int	`1`	any	seed for fill=random -- the same seed fills RAM the same way at every POWER, so a run is repeatable

`memory`

RAM/ROM board: a list of regions and PHANTOM* -- plain, unbanked memory (bank switching is its own board, `bankmem`)

`[[board.region]]` — a list you may add

Key	Kind	Legal	Meaning
`type`	enum	`ram` | `rom`	RAM, or ROM (which needs a `mount`, unless you want an empty socket)
`at`	int	`0x0` .. `0xFFFF`	Where it starts. An address: 0000, F800
`size`	int	`1` .. `65536`	How much. Decimal, and it takes a suffix: 48K, 1024, 2M
`mount`	string	text	The ROM image. A file (relative to THIS FILE), or builtin:
`relocate`	bool	`on` | `off`	Move a HEX/S-record image to `at` instead of its own record address

Board properties

Key	Kind	Default	Legal	Meaning
`honors_phantom`	enum	`all`	`none` | `read` | `all`	A JUMPER. Another board pulls PHANTOM* -- do I switch off? none | read | all
`phantom`	enum	`all`	`none` | `read` | `all`	What I ASSERT over my rom regions: none | read | all
`fill`	enum	`random`	`zero` | `random`	RAM contents at power-on: zero | random (real RAM is not zeroed)
`seed`	int	`1`	any	Seed for fill=random. The same seed fills RAM the same way at every POWER, so a run is repeatable; change it for a different junk pattern
`pages`	string	—	—	the composite page map -- which pages this board answers for. Derived from the regions you declared **(read-only — not a key you may set)**

`v2z80rom`

S100Computers V2 Z80 CPU board -- its onboard paged monitor EEPROM (the Z80 itself is board 'z80cpu'). An 8K 28C64 at F000-FFFF holding two 4K pages, builtin:master0 (low) / master1 (high), selected by OUT D3H bit1 (bit0=1 inactivates the EEPROM so RAM shows through). Shadows RAM in its window while enabled. Cold-start the MASTER monitor with startup=["RUN F000"]; the 'I' command boots CP/M 3 off a dualsd card

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0xD3`	`0x0` .. `0xFF`	Page/ROM-control latch (D3H): bit1 = EEPROM A12 page select, bit0 = ROM inactivate

Disk

`16fdc`

Cromemco 16FDC: WD FD1793 soft-sector floppy (single + double density), up to 4 drives. Disk registers at 30-34, a TMS 5501 console UART at 00-09 (unit 'tty'), and a 4K RDOS 2.52 boot PROM at C000 (OUT 40H banks it out, RESET restores it). Boots CDOS

Units: `tty` (serial, CONNECT), `drive0` (disk, MOUNT), `drive1` (disk, MOUNT), `drive2` (disk, MOUNT), `drive3` (disk, MOUNT)

`[[board.drive]]` — a list you may add

Key	Kind	Legal	Meaning
`unit`	int	`0` .. `3`	Which drive (0..3)
`mount`	string	text	The disk image to put in it. Relative to THIS FILE.
`readonly`	bool	`on` | `off`	Write-protect the disk. The drive senses it, so the guest is never told *(also `writeprotect`)*

Board properties

Key	Kind	Default	Legal	Meaning
`bootstrap`	bool	`true`	`on` | `off`	The BOOT/MON strap. On (default): the RDOS ROM is mapped at C000 and ¬BOOT reads low, so RDOS boots the disk. Off: the ROM still answers but ¬BOOT reads high (the monitor prompt instead of an auto-boot)
`boot_drive`	enum	`A`	`A` | `B` | `C` | `D`	The boot-drive switches (16FDC switches 7 and 8, 64FDC switches 3 and 4). RDOS reads them for an automatic boot, and for its B command with no drive letter
`drives`	int	`4`	`1` .. `4`	Drives on the controller (A-D, one-hot select DS4-DS1)
`timing`	enum	`full`	`full` | `real`	Disk timing. full: a disk access completes at once. real: with Auto Wait armed, IN 34 holds READY until DRQ or the end of the command, so seek, head settle and byte times take emulated time, as on the board

Unit `tty` — `[board.unit.tty]`

Key	Kind	Default	Legal	Meaning
`baud`	int	`9600`	`0` .. `76800`	Line rate. The guest sets it by writing the baud register; this seeds it and shows the effective rate (0 = the register selected no rate)
`rate`	string	`full`	text	Console speed: full (as fast as the guest reads) | real (wall-clock baud)
`dcd`	enum	`ground`	`ground` | `wired`	/DCD pin: grounded on the card, or wired to the connector
`cts`	enum	`ground`	`ground` | `wired`	/CTS pin: grounded on the card, or wired -- and then it gates the transmitter
`lines`	string	—	—	Live pin state (read-only). CAPITALS = asserted. in: DCD CTS **(read-only — not a key you may set)**
`connect`	string	`null`	text	The endpoint on the other end of the line (CONNECT sets this)

`64fdc`

Cromemco 64FDC: the 16FDC's 1983 successor -- same FD1793 + TMS 5501, carrying an 8K RDOS 3.12 boot PROM at C000-DFFF (OUT 40H banks it out, RESET restores it). Boots CDOS

Units: `tty` (serial, CONNECT), `drive0` (disk, MOUNT), `drive1` (disk, MOUNT), `drive2` (disk, MOUNT), `drive3` (disk, MOUNT)

`[[board.drive]]` — a list you may add

Key	Kind	Legal	Meaning
`unit`	int	`0` .. `3`	Which drive (0..3)
`mount`	string	text	The disk image to put in it. Relative to THIS FILE.
`readonly`	bool	`on` | `off`	Write-protect the disk. The drive senses it, so the guest is never told *(also `writeprotect`)*

Board properties

Key	Kind	Default	Legal	Meaning
`bootstrap`	bool	`true`	`on` | `off`	The BOOT/MON strap. On (default): the RDOS ROM is mapped at C000 and ¬BOOT reads low, so RDOS boots the disk. Off: the ROM still answers but ¬BOOT reads high (the monitor prompt instead of an auto-boot)
`boot_drive`	enum	`A`	`A` | `B` | `C` | `D`	The boot-drive switches (16FDC switches 7 and 8, 64FDC switches 3 and 4). RDOS reads them for an automatic boot, and for its B command with no drive letter
`drives`	int	`4`	`1` .. `4`	Drives on the controller (A-D, one-hot select DS4-DS1)
`timing`	enum	`full`	`full` | `real`	Disk timing. full: a disk access completes at once. real: with Auto Wait armed, IN 34 holds READY until DRQ or the end of the command, so seek, head settle and byte times take emulated time, as on the board

Unit `tty` — `[board.unit.tty]`

Key	Kind	Default	Legal	Meaning
`baud`	int	`9600`	`0` .. `76800`	Line rate. The guest sets it by writing the baud register; this seeds it and shows the effective rate (0 = the register selected no rate)
`rate`	string	`full`	text	Console speed: full (as fast as the guest reads) | real (wall-clock baud)
`dcd`	enum	`ground`	`ground` | `wired`	/DCD pin: grounded on the card, or wired to the connector
`cts`	enum	`ground`	`ground` | `wired`	/CTS pin: grounded on the card, or wired -- and then it gates the transmitter
`lines`	string	—	—	Live pin state (read-only). CAPITALS = asserted. in: DCD CTS **(read-only — not a key you may set)**
`connect`	string	`null`	text	The endpoint on the other end of the line (CONNECT sets this)

`dcdd`

MITS 88-DCDD: 8" hard-sector floppy, up to 16 drives. Three ports at BASE+0..2. INVERTED status bits

Units: `drive0` (disk, MOUNT), `drive1` (disk, MOUNT), `drive2` (disk, MOUNT), `drive3` (disk, MOUNT)

`[[board.drive]]` — a list you may add

Key	Kind	Legal	Meaning
`unit`	int	`0` .. `3`	Which drive on the daisy chain
`mount`	string	text	The disk image to put in it. Relative to THIS FILE.
`readonly`	bool	`on` | `off`	Write-protect the disk. The DRIVE senses it, so the guest is never told: it writes, the head is inhibited, the bytes go nowhere *(also `writeprotect`)*
`media`	enum	`8in` | `fdc8mb`	Force the format instead of probing the image's size
`create`	bool	`on` | `off`	Make the disk file (empty) if it is not there, then mount it -- a fresh disk to FORMAT. Pair with `media` to pick its geometry

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0x8`	`0x0` .. `0xFD`	Base address. The board decodes three ports: BASE+0 .. BASE+2
`drives`	int	`4`	`1` .. `16`	Drives on the daisy chain
`interrupt`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where the card's interrupt is soldered *(interrupt strap)*

`disk1`

CompuPro Disk 1 (1981), the Disk 1A's predecessor: an NEC uPD765A floppy controller for 8" drives, single and double density, up to 4 drives. Four ports at BASE+0..3 (default C0): 765 status, 765 data, board status / DMA address, and a bit-banged serial line on D7. All data moves by DMA. Carries a 2716 boot EPROM whose selected 256-byte routine answers at 0000 over PHANTOM* until the guest switches it off; any reset turns it back on

Units: `drive0` (disk, MOUNT), `drive1` (disk, MOUNT), `drive2` (disk, MOUNT), `drive3` (disk, MOUNT), `serial` (serial, CONNECT)

`[[board.drive]]` — a list you may add

Key	Kind	Legal	Meaning
`unit`	int	`0` .. `3`	Which drive (0..3)
`mount`	string	text	The disk image to put in it. Relative to THIS FILE.
`readonly`	bool	`on` | `off`	Write-protect the disk. The drive senses it, so the guest is never told *(also `writeprotect`)*
`media`	enum	`8ss` | `8ss-sd` | `8ss-dd256` | `8ss-dd512` | `8ss-dd1024` | `8ds` | `8ds-sd` | `8ds-dd256` | `8ds-dd512` | `8ds-dd1024`	The diskette: its size and sides, such as 8ds, or a whole format, such as 8ds-dd1024. A blank disk needs it. Without it, the size of the image decides
`create`	bool	`on` | `off`	Make an empty disk file if it is not there, then mount it as a blank disk. A blank needs `media`

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0xC0`	`0x0` .. `0xFC`	Base address (S2-3..8). The board decodes four ports: BASE+0 .. BASE+3 (default C0)
`rom`	string	`builtin:DISK1`	text	The boot EPROM: builtin:DISK1 (default), or a 2716 image file (Intel HEX, S-records or binary)
`routine`	int	`4`	`0` .. `7`	Boot routine (S2-1..2 and J17): the 256-byte page of the EPROM at 0000. J17 at A adds 4. In builtin:DISK1, 4-7 are 8080/Z80 code (default 4)
`boot`	bool	`true`	`on` | `off`	Boot EPROM enable (S1-4). On (default, S1-4 OFF): the ROM answers reads at 0000-00FF over PHANTOM* after a reset. Off: a plain disk controller
`port3`	enum	`serial`	`serial` | `motor`	S1-3: what the port 3 D7 latch drives. serial (default): the RS-232 output line. motor: the MOTOR ON line of the minifloppy cable, D7 = 0 on
`interrupt`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where the 765's INT goes (J0-J7 = vi0-vi7, J8 = int; default none, polled) *(interrupt strap)*

Unit `serial` — `[board.unit.serial]`

Key	Kind	Default	Legal	Meaning
`baud`	int	`9600`	`50` .. `115200`	Line rate: the far end's setting, and what the guest's delay loop must match. 8 data bits, no parity, 1 stop bit
`connect`	string	`null`	text	The endpoint on the other end of the line (CONNECT sets this)

`disk1a`

CompuPro Disk 1A: an NEC uPD765A floppy controller for 8" and 5.25" drives, single and double density, up to 4 drives. Four ports at BASE+0..3 (default C0): 765 status / drive select, 765 data, board status / DMA address, motor control. All data moves by DMA. Carries a boot EPROM whose selected 512-byte routine answers at 0000 over PHANTOM* until the guest switches it off; any reset turns it back on

Units: `drive0` (disk, MOUNT), `drive1` (disk, MOUNT), `drive2` (disk, MOUNT), `drive3` (disk, MOUNT)

`[[board.drive]]` — a list you may add

Key	Kind	Legal	Meaning
`unit`	int	`0` .. `3`	Which drive (0..3)
`mount`	string	text	The disk image to put in it. Relative to THIS FILE.
`readonly`	bool	`on` | `off`	Write-protect the disk. The drive senses it, so the guest is never told *(also `writeprotect`)*
`media`	enum	`8ss` | `8ss-sd` | `8ss-dd256` | `8ss-dd512` | `8ss-dd1024` | `8ds` | `8ds-sd` | `8ds-dd256` | `8ds-dd512` | `8ds-dd1024` | `5ss` | `5ss-sd` | `5ss-dd256` | `5ss-dd512` | `5ss-dd1024` | `5ds` | `5ds-sd` | `5ds-dd256` | `5ds-dd512` | `5ds-dd1024`	The diskette: its size and sides, such as 8ds, or a whole format, such as 8ds-dd1024. A blank disk needs it. Without it, the size of the image decides
`create`	bool	`on` | `off`	Make an empty disk file if it is not there, then mount it as a blank disk. A blank needs `media`

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0xC0`	`0x0` .. `0xFC`	Base address (S3-2..7). The board decodes four ports: BASE+0 .. BASE+3 (default C0)
`rom`	string	`builtin:DISK1A`	text	The boot EPROM: builtin:DISK1A (default), or a 2764 image file (Intel HEX, S-records or binary)
`routine`	int	`0`	`0` .. `15`	Boot routine (S1-2..5): the 512-byte page of the EPROM at 0000. 0-3 are 8080/Z80 code
`boot`	bool	`true`	`on` | `off`	Boot EPROM enable (S3-8). On (default): the ROM answers reads at 0000-01FF over PHANTOM* after a reset. Off: a plain disk controller
`sense_switch`	bool	`true`	`on` | `off`	S3-1, read in status bit 2. On (default, reads 0): the CompuPro ROM tells the OS to use a System Support 1 console. Off: an Interfacer 3/4
`mini_motor_control`	bool	`true`	`on` | `off`	Jumper J11, in the 5.25" MTR ON line. On (default, J11 installed): bit 7 of the Motor Control Register runs the 5.25" motors, and they stop about 15 seconds after the last access. Off: the 5.25" motors run all the time. The 8" motors always follow bits 4-7
`interrupt`	enum	`vi4`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where the 765's INT goes (J10; CompuPro used vi4) *(interrupt strap)*

`dualide`

S100Computers IDE-AB (CF): the IDE/CompactFlash half of the IDE+ESP32 combination board -- two CF sockets (drives 0/1 = A:/B:) for CP/M 3. Five 8255 ports at BASE+0..4 (default 30): A/B data, C control lines, mode config, drive select. Programmed-I/O ATA register engine (LBA read/write, 512-byte sectors). No boot PROM -- the CPU board's monitor boots CP/M from the CF. Mounts the SAME card image as dualsd (a .img with a .geo geometry sidecar); pair with dualsd for the full A:/B:+C:/D: system

Units: `drive0` (disk, MOUNT), `drive1` (disk, MOUNT)

`[[board.drive]]` — a list you may add

Key	Kind	Legal	Meaning
`unit`	int	`0` .. `1`	Which CF socket (0 = drive A:, 1 = drive B:)
`mount`	string	text	The card image (a .img with a sibling .geo geometry) to put in it. Relative to THIS FILE.
`readonly`	bool	`on` | `off`	Write-protect the card *(also `writeprotect`)*

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0x30`	`0x0` .. `0xFB`	Base address. The board decodes five 8255 ports: BASE..BASE+4 (A,B,C,cfg,drive)

`dualsd`

S100Computers Dual SD: two microSD sockets (drives 0/1) presented as raw 512-byte-sector CF/SD cards, for CP/M 3. Two ports at BASE+0..1 (default 80): status/command + data. Programmed-I/O command/handshake engine (33H-lead + 8 commands). No boot PROM -- the CPU board's monitor loads CP/M from track 0. Mount a card image (a .img with a .geo geometry sidecar)

Units: `drive0` (disk, MOUNT), `drive1` (disk, MOUNT)

`[[board.drive]]` — a list you may add

Key	Kind	Legal	Meaning
`unit`	int	`0` .. `1`	Which SD socket (0 = drive 1 / C:, 1 = drive 2 / D:)
`mount`	string	text	The card image (a .img with a sibling .geo geometry) to put in it. Relative to THIS FILE.
`readonly`	bool	`on` | `off`	Write-protect the card *(also `writeprotect`)*

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0x80`	`0x0` .. `0xFE`	Base address. The board decodes two ports: BASE (status/command) and BASE+1 (data)

`fdcplus`

FarmTek FDC+ in serial-drive mode: an 88-DCDD/88-MDS-compatible controller with no drive at all -- a drive server on unit 'line' holds the images and the card fetches a whole track at a time. drivetype=7 (8", incl. the 8 MB drive) or 6 (Minidisk), read at power-on. Four ports at BASE+0..3 (08 or 80)

Units: `line` (serial, CONNECT), `drive0` (disk, MOUNT), `drive1` (disk, MOUNT), `drive2` (disk, MOUNT), `drive3` (disk, MOUNT)

`[[board.drive]]` — a list you may add

Key	Kind	Legal	Meaning
`unit`	int	`0` .. `3`	Which drive: 0-3. Drive type 5 only
`mount`	string	text	The 1.5 MB disk image to put in it. Relative to THIS FILE.
`readonly`	bool	`on` | `off`	Write-protect the disk. The drive tells the card, and the card tells the 8080: status bit 4 *(also `writeprotect`)*

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0x8`	`0x0` .. `0xFC`	Base address: 08, or 80 with the address jumpers moved. Four ports, BASE+0 .. BASE+3
`drivetype`	int	`7`	`5` .. `7`	Drive Type switches (S3), read at power-on: 5 = 1.5 MB floppy, 6 = serial drive as a Minidisk, 7 = serial drive as an 8" drive
`baud`	int	`403200`	`9600` .. `460800`	Serial drive baud rate: 9600, 19200, 38400, 57600, 76800, 230400, 403200 (preferred) or 460800
`connect`	string	`null`	text	The drive server on the other end of the line (CONNECT sets this)

`hdsk`

MITS 88-HDSK Datakeeper: Pertec hard disk, 256-byte sectors from a linear .DSK. Eight ports at BASE+0..7 (default A0). Command/handshake protocol, four page buffers

Units: `drive0` (disk, MOUNT)

`[[board.drive]]` — a list you may add

Key	Kind	Legal	Meaning
`unit`	int	any	Which logical drive (slot = unit*2 + platter)
`mount`	string	text	The disk image to put in it. Relative to THIS FILE.
`readonly`	bool	`on` | `off`	Write-protect the disk *(also `writeprotect`)*

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0xA0`	`0x0` .. `0xF8`	Base address. The board decodes eight ports: BASE+0 .. BASE+7
`drives`	int	`1`	`1` .. `8`	Logical drives (one platter each): slot = unit*2 + platter
`interrupt`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where the card's interrupt is soldered *(interrupt strap)*

`icom`

iCOM FD3712/FD3812 8" floppy: a programmed-I/O command/handshake controller on the S-100 Interface board. Two ports at BASE+0..1 (default C0) plus a boot PROM and 6810 scratch RAM in high memory (rom=builtin:icom-fd3712-cpm | icom-fd3712-fdos | icom-fd3812-cpm). Boots CP/M 2.2 (single and double density) and FDOS. Up to 4 drives

Units: `drive0` (disk, MOUNT), `drive1` (disk, MOUNT)

`[[board.drive]]` — a list you may add

Key	Kind	Legal	Meaning
`unit`	int	`0` .. `1`	Which drive (0..3)
`mount`	string	text	The disk image to put in it. Relative to THIS FILE.
`readonly`	bool	`on` | `off`	Write-protect the disk *(also `writeprotect`)*

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0xC0`	`0x0` .. `0xFE`	Base address. The board decodes two ports: BASE (command/status) and BASE+1 (data)
`rom`	string	`builtin:icom-fd3712-cpm`	text	The boot PROM this interface board carries: builtin:icom-fd3712-cpm (CP/M 2.2, single density, F000), builtin:icom-fd3712-fdos (FDOS, C000), or builtin:icom-fd3812-cpm (CP/M 2.2, double density, F000). The window base and the 6810 scratch RAM follow the image
`drives`	int	`2`	`1` .. `4`	Drives on the controller (unit 0..3, selected by cDRVSEC bits 7:6)

`mds`

MITS 88-MDS: 5.25" minidisk, 4 drives. Same three ports as the dcdd -- but 300 RPM, 64 us/byte, and a motor that stops after 6.4 s

Units: `drive0` (disk, MOUNT), `drive1` (disk, MOUNT), `drive2` (disk, MOUNT), `drive3` (disk, MOUNT)

`[[board.drive]]` — a list you may add

Key	Kind	Legal	Meaning
`unit`	int	`0` .. `3`	Which drive on the daisy chain
`mount`	string	text	The disk image to put in it. Relative to THIS FILE.
`readonly`	bool	`on` | `off`	Write-protect the disk. The DRIVE senses it, so the guest is never told: it writes, the head is inhibited, the bytes go nowhere *(also `writeprotect`)*
`media`	enum	`minidisk`	Force the format instead of probing the image's size
`create`	bool	`on` | `off`	Make the disk file (empty) if it is not there, then mount it -- a fresh disk to FORMAT. Pair with `media` to pick its geometry

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0x8`	`0x0` .. `0xFD`	Base address. The board decodes three ports: BASE+0 .. BASE+2
`drives`	int	`4`	`1` .. `4`	Drives on the daisy chain
`interrupt`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where the card's interrupt is soldered *(interrupt strap)*
`motor`	enum	`free`	`free` | `real`	free: always at speed (default). real: 1 s spin-up, and it stops after 6.4 s

`mdsa`

North Star MDS-A: single-density 5.25" hard-sector floppy, up to 3 drives. No ports: a 1 K memory block at BASE (default E800) where a memory READ is the command. Boot PROM on the board -- RUN E900

Units: `drive0` (disk, MOUNT), `drive1` (disk, MOUNT), `drive2` (disk, MOUNT)

`[[board.drive]]` — a list you may add

Key	Kind	Legal	Meaning
`unit`	int	`0` .. `2`	Which drive. Unit 0 is the drive that North Star calls drive 1
`mount`	string	text	The disk image to put in it. Relative to THIS FILE.
`readonly`	bool	`on` | `off`	Write-protect the diskette. The guest sees it in the WP status bit *(also `writeprotect`)*
`media`	enum	`sd`	Force the format instead of probing the image's size
`create`	bool	`on` | `off`	Make the disk file (empty) if it is not there, then mount it -- a blank diskette for the guest to format

Board properties

Key	Kind	Default	Legal	Meaning
`base`	int	`0xE800`	`0x0` .. `0xFC00`	Origin of the 1 K block. The built-in PROM is the standard part and runs only at E800
`drives`	int	`3`	`1` .. `3`	Drives on the cable
`interrupt`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where the sector-pulse interrupt is jumpered (lower left of the board) *(interrupt strap)*
`motor`	enum	`free`	`free` | `real`	free: the motors run until they are stopped (default). real: they stop by themselves after the board's count of revolutions
`timing`	enum	`full`	`full` | `real`	Disk timing. full: a read-data or write-data access completes at once. real: it holds READY until the byte is under the head, so a sector transfer takes emulated time, as on the board

`mdsad`

North Star MDS-A-D: double-density 5.25" hard-sector floppy (single, double and two-sided), up to 4 drives. No ports: a 1 K memory block at BASE (default E800) where a memory READ is the command. Boot PROM on the board -- RUN E800

Units: `drive0` (disk, MOUNT), `drive1` (disk, MOUNT), `drive2` (disk, MOUNT), `drive3` (disk, MOUNT)

`[[board.drive]]` — a list you may add

Key	Kind	Legal	Meaning
`unit`	int	`0` .. `3`	Which drive. Unit 0 is the drive that North Star calls drive 1
`mount`	string	text	The disk image to put in it. Relative to THIS FILE.
`readonly`	bool	`on` | `off`	Write-protect the diskette. The guest sees it in the WP status bit *(also `writeprotect`)*
`media`	enum	`sd` | `dd` | `quad`	Force the format instead of probing the image's size
`create`	bool	`on` | `off`	Make the disk file (empty) if it is not there, then mount it -- a blank diskette for the guest to format

Board properties

Key	Kind	Default	Legal	Meaning
`base`	int	`0xE800`	`0x0` .. `0xFC00`	Origin of the 1 K block. The built-in PROM is the standard part and runs only at E800
`drives`	int	`4`	`1` .. `4`	Drives on the cable
`interrupt`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where the sector-pulse interrupt is jumpered (lower left of the board) *(interrupt strap)*
`motor`	enum	`free`	`free` | `real`	free: the motors run until they are stopped (default). real: they stop by themselves after the board's count of revolutions
`timing`	enum	`full`	`full` | `real`	Disk timing. full: a read-data or write-data access completes at once. real: it holds READY until the byte is under the head, so a sector transfer takes emulated time, as on the board

`tarbell`

Tarbell #1011: single-density WD FD1771 floppy, up to 4 drives. Eight ports at BASE+0..7 (default F8). Carries a 32-byte boot PROM that shadows 0000 over PHANTOM* -- boots CP/M automatically at reset (bootstrap=on)

Units: `drive0` (disk, MOUNT), `drive1` (disk, MOUNT), `drive2` (disk, MOUNT), `drive3` (disk, MOUNT)

`[[board.drive]]` — a list you may add

Key	Kind	Legal	Meaning
`unit`	int	`0` .. `3`	Which drive (0..3)
`mount`	string	text	The disk image to put in it. Relative to THIS FILE.
`readonly`	bool	`on` | `off`	Write-protect the disk. The drive senses it, so the guest is never told *(also `writeprotect`)*

Board properties

Key	Kind	Default	Legal	Meaning
`bootstrap`	bool	`true`	`on` | `off`	The boot-PROM enable DIP. On (default): the 32-byte PROM shadows 0000 over PHANTOM* at reset and boots the disk. Off: a plain disk controller, no PROM
`port`	int	`0xF8`	`0x0` .. `0xF8`	Base address. The board decodes eight ports: BASE+0 .. BASE+7 (default F8)
`drives`	int	`4`	`1` .. `4`	Drives on the controller (binary select 0-3)
`interrupt`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where the card's interrupt is soldered *(interrupt strap)*
`timing`	enum	`full`	`full` | `real`	Disk timing. full: a disk access completes at once. real: the wait port holds READY until the next byte or the end of the command, so seek, head settle and byte times take emulated time, as on the board

`tarbelldd`

Tarbell #2022: double-density WD FD1791 floppy (mixed-density media, SD track 0), up to 4 drives. The single-density card's twin with a bitmap OUT-FC latch and a port-FD DMA/ext-addr register. Same 32-byte boot PROM

Units: `drive0` (disk, MOUNT), `drive1` (disk, MOUNT), `drive2` (disk, MOUNT), `drive3` (disk, MOUNT)

`[[board.drive]]` — a list you may add

Key	Kind	Legal	Meaning
`unit`	int	`0` .. `3`	Which drive (0..3)
`mount`	string	text	The disk image to put in it. Relative to THIS FILE.
`readonly`	bool	`on` | `off`	Write-protect the disk. The drive senses it, so the guest is never told *(also `writeprotect`)*

Board properties

Key	Kind	Default	Legal	Meaning
`bootstrap`	bool	`true`	`on` | `off`	The boot-PROM enable DIP. On (default): the 32-byte PROM shadows 0000 over PHANTOM* at reset and boots the disk. Off: a plain disk controller, no PROM
`port`	int	`0xF8`	`0x0` .. `0xF8`	Base address. The board decodes eight ports: BASE+0 .. BASE+7 (default F8)
`drives`	int	`4`	`1` .. `4`	Drives on the controller (binary select 0-3)
`interrupt`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where the card's interrupt is soldered *(interrupt strap)*
`timing`	enum	`full`	`full` | `real`	Disk timing. full: a disk access completes at once. real: the wait port holds READY until the next byte or the end of the command, so seek, head settle and byte times take emulated time, as on the board
`dmaport`	int	`0xE0`	`0x0` .. `0xF0`	Base of the on-card 8257 DMA controller's 16-port register block (default E0). The DMA-mode CBIOS programs ADR/WCT here; the SD2DD strap is E0

`versafloppy`

SD Systems VersaFloppy I/II: WD FD177x soft-sector floppy, up to 4 drives. Eight ports at BASE+0..7 (default 60). variant=vfi (FD1771, single density) | vfii (FD1791, single+double). Boots SDOS with the SBC-200 + DDBIOS

Units: `drive0` (disk, MOUNT), `drive1` (disk, MOUNT), `drive2` (disk, MOUNT), `drive3` (disk, MOUNT)

`[[board.drive]]` — a list you may add

Key	Kind	Legal	Meaning
`unit`	int	`0` .. `3`	Which drive (0..3)
`mount`	string	text	The disk image to put in it. Relative to THIS FILE.
`readonly`	bool	`on` | `off`	Write-protect the disk. The drive senses it, so the guest is never told *(also `writeprotect`)*
`media`	enum	`8sd` | `8dd` | `8dd256` | `8sd-ds` | `8dd-ds` | `8dd256-ds` | `5sd` | `5sd-ds` | `5dd` | `5dd-ds`	Force the format instead of probing the image's size

Board properties

Key	Kind	Default	Legal	Meaning
`variant`	enum	`vfii`	`vfi` | `vfii`	Which board: vfi (FD1771, single density) or vfii (FD1791, single and double density). vfii is the default -- it boots SDOS's DD-256 disks
`port`	int	`0x60`	`0x0` .. `0xF8`	Base address. The board decodes eight ports: BASE+0 .. BASE+7 (60H)
`drives`	int	`4`	`1` .. `4`	Drives on the controller (one-hot select D0-D3)
`interrupt`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where the card's interrupt is soldered *(interrupt strap)*
`timing`	enum	`full`	`full` | `real`	Disk timing. full: a disk access completes at once. real: the data port holds READY until the next byte or the end of the command (while the wait-state circuit is enabled), so seek, head settle and byte times take emulated time, as on the board

Serial

`2sio`

MITS 88-2SIO: two 6850 ACIAs, units 'a' and 'b'. Four ports at BASE+0..3

Units: `a` (serial, CONNECT), `b` (serial, CONNECT)

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0x10`	`0x0` .. `0xFC`	Base address. The board decodes four ports: BASE+0 .. BASE+3

Unit `a` — `[board.unit.a]`

Key	Kind	Default	Legal	Meaning
`baud`	int	`9600`	`50` .. `76800`	Line rate. A JUMPER on the real card -- software cannot change it, and there is no free-running setting: the rate paces the line
`interrupt`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where this channel's IRQ is jumpered: none | int | vi0..vi7 *(interrupt strap)*
`dcd`	enum	`ground`	`ground` | `wired`	/DCD pin: grounded on the card, or wired to the connector
`cts`	enum	`ground`	`ground` | `wired`	/CTS pin: grounded on the card, or wired -- and then it gates the transmitter
`lines`	string	—	—	Live pin state (read-only). CAPITALS = asserted. in: DCD CTS, out: RTS BRK **(read-only — not a key you may set)**
`connect`	string	`null`	text	The endpoint on the other end of the line (CONNECT sets this)

Unit `b` — `[board.unit.b]`

Key	Kind	Default	Legal	Meaning
`baud`	int	`9600`	`50` .. `76800`	Line rate. A JUMPER on the real card -- software cannot change it, and there is no free-running setting: the rate paces the line
`interrupt`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where this channel's IRQ is jumpered: none | int | vi0..vi7 *(interrupt strap)*
`dcd`	enum	`ground`	`ground` | `wired`	/DCD pin: grounded on the card, or wired to the connector
`cts`	enum	`ground`	`ground` | `wired`	/CTS pin: grounded on the card, or wired -- and then it gates the transmitter
`lines`	string	—	—	Live pin state (read-only). CAPITALS = asserted. in: DCD CTS, out: RTS BRK **(read-only — not a key you may set)**
`connect`	string	`null`	text	The endpoint on the other end of the line (CONNECT sets this)

`gsio`

Generic SIO: a strap-configurable serial board with TWO independent channels, units 'a' and 'b' (configure each under [board.unit.a] / [board.unit.b]). Basic transmit/receive only -- no programmable word length, parity, stop bits or framing/overrun status; a specific card that needs those is a separate emulated board. Per channel you strap: a status/control port (status_port -- read synthesizes DAV/TBMT at bit positions you pick, write is discarded) and a data port (data_port); one inverter_gate knob inverts both status bits together. Built-in profiles preset the straps: profile=sior1 (MITS SIO Rev 1, the default) | sior0 (MITS SIO Rev 0) | tuart (Cromemco TU-ART) | imsai-sio2 | compupro-if2 (CompuPro Interfacer II) | compupro-ss1 (CompuPro System Support 1). Channel a defaults to ports 0/1, b to 2/3. Polled, no interrupts. CONNECT each channel to a file, socket, serial port, in:/out:

Units: `a` (serial, CONNECT), `b` (serial, CONNECT)

Board properties

No settable properties.

Unit `a` — `[board.unit.a]`

Key	Kind	Default	Legal	Meaning
`profile`	enum	`sior1`	`custom` | `sior1` | `sior0` | `tuart` | `imsai-sio2` | `compupro-if2` | `compupro-ss1`	Built-in card to preset the straps from: custom, or a named board. Selecting one sets status_port/data_port/bits/inverter_gate (still overridable)
`status_port`	int	`0x0`	`0x0` .. `0xFF`	Status(read)/control(write) port. Control writes are discarded
`data_port`	int	`0x1`	`0x0` .. `0xFF`	Data port: receive(read)/transmit(write)
`dav`	int	`0`	`0` .. `7`	Status bit (0-7) carrying DAV, data available (a byte to receive)
`tbmt`	int	`7`	`0` .. `7`	Status bit (0-7) carrying TBMT, transmit buffer empty (ready to send)
`inverter_gate`	bool	`true`	`on` | `off`	Route both status bits through the inverter gate -- asserted DAV/TBMT read 0 (active low)
`baud`	int	`9600`	any	Line rate programmed onto a CONNECTed real serial port (8N1). Inert on a socket/file; does not pace the emulated line
`connect`	string	`null`	text	The endpoint on the serial line (CONNECT sets this): a file, socket, serial port, in:/out: file, null, loopback

Unit `b` — `[board.unit.b]`

Key	Kind	Default	Legal	Meaning
`profile`	enum	`sior1`	`custom` | `sior1` | `sior0` | `tuart` | `imsai-sio2` | `compupro-if2` | `compupro-ss1`	Built-in card to preset the straps from: custom, or a named board. Selecting one sets status_port/data_port/bits/inverter_gate (still overridable)
`status_port`	int	`0x2`	`0x0` .. `0xFF`	Status(read)/control(write) port. Control writes are discarded
`data_port`	int	`0x3`	`0x0` .. `0xFF`	Data port: receive(read)/transmit(write)
`dav`	int	`0`	`0` .. `7`	Status bit (0-7) carrying DAV, data available (a byte to receive)
`tbmt`	int	`7`	`0` .. `7`	Status bit (0-7) carrying TBMT, transmit buffer empty (ready to send)
`inverter_gate`	bool	`true`	`on` | `off`	Route both status bits through the inverter gate -- asserted DAV/TBMT read 0 (active low)
`baud`	int	`9600`	any	Line rate programmed onto a CONNECTed real serial port (8N1). Inert on a socket/file; does not pace the emulated line
`connect`	string	`null`	text	The endpoint on the serial line (CONNECT sets this): a file, socket, serial port, in:/out: file, null, loopback

`io4`

SSM IO-4 (2P+2S): the real Solid State Music board -- two full-duplex serial channels AND a four-port parallel section. Serial units 'a'/'b' are real 1602-family UARTs with programmable word length (data_bits 5-8), parity and stop bits (unlike the generic gsio), plus the full status-word strap-up (stat_* map, invert_status, port_reversal) and named profiles (default altair-rev1). A 4-port block set by switch S3 (default 0-3): Serial A status/data at BASE+0/+1, Serial B at BASE+2/+3. Parallel units 'pa'/'pb' are 8212 latched ports (input latch + service request, output latch) on their own 2-port block set by switch S4 (par_port, default 4-5): Parallel A at PAR+0, B at PAR+1; a byte the far end sends is strobed in, a write latches out, and the §3.2.2 status/data console flag is strappable (dav_bit/dav_source/dav_active_low). If the two blocks overlap, neither section answers the shared ports. Each serial channel's receive (DAV) and transmit (TBMT) and each parallel input (service request) can raise an interrupt, strapped on header W4 to a VI line, pin 73 (int), or none (rx_int/tx_int per serial channel, int per parallel port) -- there is no software enable, the strap is the enable, and a parallel input interrupt rises even when the port is not addressed. Configure each unit under [board.unit.a] / [board.unit.pa] etc.; CONNECT each to a file, socket, serial port, in:/out:

Units: `a` (serial, CONNECT), `b` (serial, CONNECT), `pa` (serial, CONNECT), `pb` (serial, CONNECT)

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0x0`	`0x0` .. `0xFC`	Serial base address (switch S3) -- a 4-PORT BLOCK, so a multiple of 4. Serial A at BASE+0/+1, Serial B at BASE+2/+3
`par_port`	int	`0x4`	`0x0` .. `0xFE`	Parallel base address (switch S4) -- a 2-PORT BLOCK, so a multiple of 2. Parallel A (J6-in/J5-out) at BASE, Parallel B (J4-in/J3-out) at BASE+1. If it overlaps the serial block, neither section answers the shared ports

Unit `a` — `[board.unit.a]`

Key	Kind	Default	Legal	Meaning
`profile`	enum	`altair-rev1`	`custom` | `altair-rev1` | `altair-rev0` | `i8251` | `proctech` | `imsai`	Preset the status straps from a host personality: custom, altair-rev1 (the default and the SSM 8080 monitor console), altair-rev0, i8251, proctech, imsai. Sets stat_*, invert_status and port_reversal (overridable)
`stat_dav`	enum	`0`	`none` | `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`	Data-bus bit carrying DAV, data available (0-7 | none)
`stat_tbmt`	enum	`7`	`none` | `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`	Data-bus bit carrying TBMT, transmit buffer empty (0-7 | none)
`stat_teoc`	enum	`none`	`none` | `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`	Data-bus bit carrying TEOC, transmitter end of character (0-7 | none)
`stat_ror`	enum	`none`	`none` | `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`	Data-bus bit carrying ROR, receiver over-run (0-7 | none; always inactive)
`stat_rpe`	enum	`none`	`none` | `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`	Data-bus bit carrying RPE, receiver parity error (0-7 | none; always inactive)
`stat_rfe`	enum	`none`	`none` | `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`	Data-bus bit carrying RFE, receiver framing error (0-7 | none; always inactive)
`invert_status`	bool	`true`	`on` | `off`	Invert every status bit -- a 74368 buffer (negative sense): an asserted signal reads 0, an inactive one reads 1. Off = 74367 (positive sense)
`port_reversal`	bool	`false`	`on` | `off`	Reverse the channel's two ports (S1/S2-PR): off = status first, data last (MITS, Proc Tech); on = data first, status last (IMSAI)
`rx_int`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where the receive interrupt (a new character, DAV) is strapped on header W4: none | int | vi0..vi7. Canonical: vi1 (Serial A) / vi0 (Serial B) *(interrupt strap)*
`tx_int`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where the transmit interrupt (buffer empty, TBMT) is strapped on header W4: none | int | vi0..vi7. It is a LEVEL -- held while the transmitter is idle. Canonical: vi2 (Serial A) / vi3 (Serial B) *(interrupt strap)*
`baud`	int	`9600`	`50` .. `25000`	Line rate (header W3). RX and TX share one rate here -- a real host serial port cannot be split. Canonical IO-4 rates: 55-9600
`data_bits`	int	`8`	`5` .. `8`	Data bits per character (S1/S2 NDB1+NDB2)
`stop_bits`	int	`1`	`1` .. `2`	Stop bits (S1/S2 NSB): 1 or 2

`parity`	enum	`none`	`none` | `odd` | `even`	Parity (S1/S2 NPB/POE): none | odd | even
`connect`	string	`null`	text	The endpoint on this channel's line (CONNECT sets this): a file, socket, serial port, in:/out: file, null, loopback

Unit `b` — `[board.unit.b]`

Key	Kind	Default	Legal	Meaning
`profile`	enum	`altair-rev1`	`custom` | `altair-rev1` | `altair-rev0` | `i8251` | `proctech` | `imsai`	Preset the status straps from a host personality: custom, altair-rev1 (the default and the SSM 8080 monitor console), altair-rev0, i8251, proctech, imsai. Sets stat_*, invert_status and port_reversal (overridable)
`stat_dav`	enum	`0`	`none` | `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`	Data-bus bit carrying DAV, data available (0-7 | none)
`stat_tbmt`	enum	`7`	`none` | `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`	Data-bus bit carrying TBMT, transmit buffer empty (0-7 | none)
`stat_teoc`	enum	`none`	`none` | `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`	Data-bus bit carrying TEOC, transmitter end of character (0-7 | none)
`stat_ror`	enum	`none`	`none` | `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`	Data-bus bit carrying ROR, receiver over-run (0-7 | none; always inactive)
`stat_rpe`	enum	`none`	`none` | `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`	Data-bus bit carrying RPE, receiver parity error (0-7 | none; always inactive)
`stat_rfe`	enum	`none`	`none` | `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`	Data-bus bit carrying RFE, receiver framing error (0-7 | none; always inactive)
`invert_status`	bool	`true`	`on` | `off`	Invert every status bit -- a 74368 buffer (negative sense): an asserted signal reads 0, an inactive one reads 1. Off = 74367 (positive sense)
`port_reversal`	bool	`false`	`on` | `off`	Reverse the channel's two ports (S1/S2-PR): off = status first, data last (MITS, Proc Tech); on = data first, status last (IMSAI)
`rx_int`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where the receive interrupt (a new character, DAV) is strapped on header W4: none | int | vi0..vi7. Canonical: vi1 (Serial A) / vi0 (Serial B) *(interrupt strap)*
`tx_int`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where the transmit interrupt (buffer empty, TBMT) is strapped on header W4: none | int | vi0..vi7. It is a LEVEL -- held while the transmitter is idle. Canonical: vi2 (Serial A) / vi3 (Serial B) *(interrupt strap)*
`baud`	int	`9600`	`50` .. `25000`	Line rate (header W3). RX and TX share one rate here -- a real host serial port cannot be split. Canonical IO-4 rates: 55-9600
`data_bits`	int	`8`	`5` .. `8`	Data bits per character (S1/S2 NDB1+NDB2)
`stop_bits`	int	`1`	`1` .. `2`	Stop bits (S1/S2 NSB): 1 or 2

`parity`	enum	`none`	`none` | `odd` | `even`	Parity (S1/S2 NPB/POE): none | odd | even
`connect`	string	`null`	text	The endpoint on this channel's line (CONNECT sets this): a file, socket, serial port, in:/out: file, null, loopback

Unit `pa` — `[board.unit.pa]`

Key	Kind	Default	Legal	Meaning
`connect`	string	`null`	text	The endpoint on this parallel port's line (CONNECT sets this): a file, socket, printer, in:/out: file, null. A WRITE latches out; a byte the far end sends is strobed into the input latch
`dav_bit`	enum	`none`	`none` | `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`	Data-bus bit a data-available flag is jumpered onto when this port is read (§3.2.2 status/data console): 0-7, or none
`dav_source`	enum	`self`	`self` | `sibling`	Whose service request the dav_bit shows: self (this port's) or sibling (the other parallel port's -- the status/data console reads the data port's flag at the status port)
`dav_active_low`	bool	`false`	`on` | `off`	Present the data-available flag active-low -- the dav_bit reads 0 when a byte is waiting (§3.2.2 "D0 going low"). Off = active-high
`int`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where this parallel input's interrupt (a byte latched in, service request) is strapped on header W4: none | int | vi0..vi7. Canonical: vi6 (Parallel A) / vi5 (Parallel B) *(interrupt strap)*

Unit `pb` — `[board.unit.pb]`

Key	Kind	Default	Legal	Meaning
`connect`	string	`null`	text	The endpoint on this parallel port's line (CONNECT sets this): a file, socket, printer, in:/out: file, null. A WRITE latches out; a byte the far end sends is strobed into the input latch
`dav_bit`	enum	`none`	`none` | `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`	Data-bus bit a data-available flag is jumpered onto when this port is read (§3.2.2 status/data console): 0-7, or none
`dav_source`	enum	`self`	`self` | `sibling`	Whose service request the dav_bit shows: self (this port's) or sibling (the other parallel port's -- the status/data console reads the data port's flag at the status port)
`dav_active_low`	bool	`false`	`on` | `off`	Present the data-available flag active-low -- the dav_bit reads 0 when a byte is waiting (§3.2.2 "D0 going low"). Off = active-high
`int`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where this parallel input's interrupt (a byte latched in, service request) is strapped on header W4: none | int | vi0..vi7. Canonical: vi6 (Parallel A) / vi5 (Parallel B) *(interrupt strap)*

`pmmi`

PMMI MM-103: Bell 103 modem on an S-100 card, unit 'line'. Four ports at BASE+0..3 (default C0), read/write different registers. dial=host:port places a call over TCP when the guest goes off-hook with DTR; answer=port rings the guest on an inbound call. CONNECT the line to a file, socket or real serial port instead. Pulse digits are not decoded; no interrupts

Units: `line` (serial, CONNECT)

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0xC0`	`0x0` .. `0xFC`	Base address -- the 6-position DIP. Four ports at BASE..BASE+3; MUST be on a 4-port boundary. Default C0 (North Star alternative E0)
`connect`	string	`null`	text	The endpoint on the phone line (CONNECT sets this). in:/out: a file, ...
`dial`	string		text	Originate target host:port (empty = cannot dial). SH off-hook + DTR dials it; the guest's pulse digits are not decoded
`answer`	string		text	Auto-answer TCP port (empty/0 = will not answer). DTR arms the listener; an inbound call RINGS until the guest answers with RI
`rtsdtr`	bool	`false`	`on` | `off`	Mirror DTR onto RTS on a CONNECTed serial port (for cables that need RTS asserted to pass data). Default off
`telnet`	bool	`true`	`on` | `off`	Speak the Telnet protocol on a dial=/answer= line, for a human telnet client (no double echo, character-at-a-time). Default on; telnet=off for a raw pipe to another simulator
`frame`	string	—	—	Live UART frame (read-only), e.g. 8N1. Set by OUT BA+0 bits 2-6 **(read-only — not a key you may set)**
`baud`	int	—	—	Live line rate (read-only), 250000/(16*N) from the OUT BA+2 divisor **(read-only — not a key you may set)**
`uart`	string	—	—	Live UART status (read-only). CAPITALS = asserted: TBMT DAV **(read-only — not a key you may set)**
`lines`	string	—	—	Live modem lines (read-only). CAPITALS = asserted: SH RI DTR ST DT RING CTS AP (DT/RING/CTS/AP from the phone line + handshake state machine) **(read-only — not a key you may set)**

`propio`

S100Computers Console IO Board (Parallax-Propeller console), unit 'serial'. A strap-configurable serial subtype preset to the board's documented convention: status/data at 00/01, RX-ready = status bit 1, TX-ready = status bit 2, both active high. Every strap (status_port/data_port/dav/tbmt/inverter_gate) is still overridable -- the real board is jumpered. Polled, no interrupts. CONNECT it to a file, socket, serial port, in:/out:

Units: `serial` (serial, CONNECT)

Board properties

Key	Kind	Default	Legal	Meaning
`profile`	enum	`custom`	`custom` | `sior1` | `sior0` | `tuart` | `imsai-sio2` | `compupro-if2` | `compupro-ss1`	Built-in card to preset the straps from: custom, or a named board. Selecting one sets status_port/data_port/bits/inverter_gate (still overridable)
`status_port`	int	`0x0`	`0x0` .. `0xFF`	Status(read)/control(write) port. Control writes are discarded
`data_port`	int	`0x1`	`0x0` .. `0xFF`	Data port: receive(read)/transmit(write)
`dav`	int	`1`	`0` .. `7`	Status bit (0-7) carrying DAV, data available (a byte to receive)
`tbmt`	int	`2`	`0` .. `7`	Status bit (0-7) carrying TBMT, transmit buffer empty (ready to send)
`inverter_gate`	bool	`false`	`on` | `off`	Route both status bits through the inverter gate -- asserted DAV/TBMT read 0 (active low)
`baud`	int	`9600`	any	Line rate programmed onto a CONNECTed real serial port (8N1). Inert on a socket/file; does not pace the emulated line
`connect`	string	`null`	text	The endpoint on the serial line (CONNECT sets this): a file, socket, serial port, in:/out: file, null, loopback

`sbc`

SD Systems SBC-100/200: Z80 single-board computer. One 8-port block (78-7F): Intel 8251 console (unit 'tty', data 7C / status 7D, RxD->/DSR auto-baud for MSMONR21), Z80-CTC (78-7B) whose ch1 raises a mode-2 keyboard interrupt (vector 0x82) off the 8251 RxRDY, and a parallel port (7E/7F) whose OUT 7F bit 1 switches the onboard PROM out. Optional onboard boot PROM via [[board.socket]] (at+mount). variant=sbc100|sbc200

Units: `tty` (serial, CONNECT)

`[[board.socket]]` — a list you may add

Key	Kind	Legal	Meaning
`at`	int	any	Where the socket sits: a socket address of the bank (etch: E000 = monitor, F000 = disk BIOS)
`mount`	string	text	What is in the socket: builtin: or a HEX/BIN path. Relative to THIS FILE.

Board properties

Key	Kind	Default	Legal	Meaning
`variant`	enum	`sbc200`	`sbc100` | `sbc200`	Which board: sbc100 (2.4576 MHz) or sbc200 (4 MHz). The console, CTC and PROM behave alike here; the CPU crystal is set on the z80 card
`rxd2dsr`	bool	`true`	`on` | `off`	RxD strapped to /DSR (the SBC auto-baud jumper). Off = a plain 8251 /DSR
`port`	int	`0x7C`	`0x0` .. `0xFE`	Base I/O address (a card jumper). Data at BASE, status/command at BASE+1. The etch default is 7C
`rom_size`	enum	`2K`	`1K` | `2K` | `4K` | `8K`	Size of each PROM socket (the X1 jumpers): 1K, 2K, 4K or 8K. The etch is 2K
`bank`	int	`3`	`0` .. `7`	Which bank the onboard memory is in (the X1 jumpers). A bank is eight sockets' worth: 0-7 for 1K, 0-3 for 2K, 0-1 for 4K, 0 for 8K. The etch is 3 (C000-FFFF)
`ram`	bool	`true`	`on` | `off`	The onboard 1K static RAM is jumpered in (X3-15 to X3-16). It is the last socket's worth of the bank: F800-FFFF on the etch
`start`	int	`0x0`	`0x0` .. `0xF000`	Auto-start address (the X16/X17/X18 jumpers): after a reset the Z80 reads the PROM here. A multiple of 1000; 0000 = no auto-start

Unit `tty` — `[board.unit.tty]`

Key	Kind	Default	Legal	Meaning
`baud`	int	`9600`	`50` .. `76800`	Line rate. On the SBC the CTC generates it; here it paces the receive line and sizes the auto-baud bit. No free-running setting (min 50)
`interrupt`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where this port's IRQ is jumpered: none | int | vi0..vi7 (decoded; not yet honored) *(interrupt strap)*
`connect`	string	`null`	text	The endpoint on the other end of the line (CONNECT sets this)

`sio`

MITS 88-SIO: one COM2502 UART, unit 'tty'. Two ports at BASE+0..1. INVERTED status bits

Units: `tty` (serial, CONNECT)

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0x0`	`0x0` .. `0xFE`	Base address -- MUST BE EVEN. Control at BASE, data at BASE+1
`rev`	enum	`1`	`0` | `1`	Board revision. 1 = the factory errata mod: ready is bit 7 (out) and bit 0 (in), both inverted. 0 = as shipped, which also reports them true-sense on bits 5 and 1
`baud`	int	`9600`	`50` .. `25000`	Line rate. A JUMPER on the real card -- software cannot change it
`data_bits`	int	`8`	`5` .. `8`	Data bits per character. The NDB1/NDB2 pads
`stop_bits`	int	`1`	`1` .. `2`	Stop bits. The NSB pad: GND = 1, +V = 2
`parity`	enum	`none`	`none` | `odd` | `even`	The NPB/POE pads: none | odd | even
`in_int`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where the IN pad is soldered (RX): none | int | vi0..vi7 *(interrupt strap)*
`out_int`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where the OUT pad is soldered (TX): none | int | vi0..vi7 *(interrupt strap)*
`connect`	string	`null`	text	The endpoint on the other end of the line (CONNECT sets this)

`turnkey`

MITS 8800b Turnkey Module: phantom boot PROM (FC00-FFFF), integrated 6850 SIO (unit 'tty', default 0x10), sense switches at FF, and the Auto-Start JMP jam. Sockets via [[board.socket]]

Units: `tty` (serial, CONNECT)

`[[board.socket]]` — a list you may add

Key	Kind	Legal	Meaning
`at`	int	any	Where the socket sits in the window (FC00/FD00/FE00/FF00)
`mount`	string	text	What is in the socket: builtin: or a HEX/BIN path. Relative to THIS FILE.

Board properties

Key	Kind	Default	Legal	Meaning
`prom`	int	`0xFC00`	`0x0` .. `0xFC00`	PROM ADDR switches: base of the 1K boot-PROM window (FC00-FFFF normal)
`start`	int	`0xFC00`	`0x0` .. `0xFF00`	START ADDR switches: Auto-Start jams JMP here at reset (a multiple of 256)
`sense`	int	`0x0`	`0x0` .. `0xFF`	Sense switches (SW6/SW7), read at port FF
`sio_base`	int	`0x10`	`0x0` .. `0xFE`	Base address of the integrated 6850 SIO (0x10 = 2SIO Port A)

Unit `tty` — `[board.unit.tty]`

Key	Kind	Default	Legal	Meaning
`baud`	int	`9600`	`50` .. `76800`	Line rate. A JUMPER on the real card -- software cannot change it, and there is no free-running setting: the rate paces the line
`interrupt`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where this channel's IRQ is jumpered: none | int | vi0..vi7 *(interrupt strap)*
`dcd`	enum	`ground`	`ground` | `wired`	/DCD pin: grounded on the card, or wired to the connector
`cts`	enum	`ground`	`ground` | `wired`	/CTS pin: grounded on the card, or wired -- and then it gates the transmitter
`lines`	string	—	—	Live pin state (read-only). CAPITALS = asserted. in: DCD CTS, out: RTS BRK **(read-only — not a key you may set)**
`connect`	string	`null`	text	The endpoint on the other end of the line (CONNECT sets this)

Tape

`acr`

MITS 88-ACR: cassette. An 88-SIO B + an FSK modem, unit 'tape'. Brings the WIND/REWIND/EXTRACT verbs and a tape counter

Units: `tape` (tape, MOUNT)

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0x6`	`0x0` .. `0xFE`	Base address -- MUST BE EVEN. Control at BASE, data at BASE+1
`rev`	enum	`1`	`0` | `1`	Board revision. 1 = the factory errata mod: ready is bit 7 (out) and bit 0 (in), both inverted. 0 = as shipped, which also reports them true-sense on bits 5 and 1
`baud`	int	`300`	`50` .. `25000`	Line rate. A JUMPER on the real card -- software cannot change it
`data_bits`	int	`8`	`5` .. `8`	Data bits per character. The NDB1/NDB2 pads
`stop_bits`	int	`1`	`1` .. `2`	Stop bits. The NSB pad: GND = 1, +V = 2
`parity`	enum	`none`	`none` | `odd` | `even`	The NPB/POE pads: none | odd | even
`in_int`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where the IN pad is soldered (RX): none | int | vi0..vi7 *(interrupt strap)*
`out_int`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where the OUT pad is soldered (TX): none | int | vi0..vi7 *(interrupt strap)*

Unit `tape` — `[board.unit.tape]`

Key	Kind	Default	Legal	Meaning
`mode`	enum	`play`	`play` | `record`	Which way the bytes go: play loads from the file, record saves to it
`format`	enum	`auto`	`auto` | `raw` | `fsk300`	How to read the mounted file: auto | raw | fsk300
`leader`	int	`15`	`0` .. `120`	Seconds of idle tone before recorded data, when writing audio
`trailer`	int	`5`	`0` .. `120`	Seconds of idle tone after recorded data, when writing audio
`waveform`	enum	`square`	`square` | `sine`	Carrier shape when writing audio: square (like real hardware) | sine
`level`	int	`36`	`1` .. `100`	Recording level as a percent of full scale, when writing audio
`rate`	enum	`full`	`full` | `real`	Playback speed: full (as fast as the guest reads) | real (wall-clock baud)
`detected`	string	—	—	What the mounted tape turned out to be (empty if nothing is mounted) **(read-only — not a key you may set)**
`position`	string	—	—	Where the tape head is now: mm:ss / total (percent) -- read-only **(read-only — not a key you may set)**
`counter`	enum	`on`	`on` | `off`	Live tape counter on the console during a load: on | off
`stop`	string	`off`	off | end | mm:ss	Auto-stop playback at this time: off | end | mm:ss

`uio`

MITS 88-UIO: serial + cassette on one board. A 6850 (unit 'serial', default 0x10) and an 88-ACR cassette section (unit 'tape', default 0x06) with motor control and a SW-1 MITS/Kansas-City modulation switch. Defaults reproduce the standard 0x10 + 0x06 layout

Units: `tape` (tape, MOUNT), `serial` (serial, CONNECT)

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0x6`	`0x0` .. `0xFE`	Base address -- MUST BE EVEN. Control at BASE, data at BASE+1
`rev`	enum	`1`	`0` | `1`	Board revision. 1 = the factory errata mod: ready is bit 7 (out) and bit 0 (in), both inverted. 0 = as shipped, which also reports them true-sense on bits 5 and 1
`baud`	int	`300`	`50` .. `25000`	Line rate. A JUMPER on the real card -- software cannot change it
`data_bits`	int	`8`	`5` .. `8`	Data bits per character. The NDB1/NDB2 pads
`stop_bits`	int	`1`	`1` .. `2`	Stop bits. The NSB pad: GND = 1, +V = 2
`parity`	enum	`none`	`none` | `odd` | `even`	The NPB/POE pads: none | odd | even
`in_int`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where the IN pad is soldered (RX): none | int | vi0..vi7 *(interrupt strap)*
`out_int`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where the OUT pad is soldered (TX): none | int | vi0..vi7 *(interrupt strap)*
`serial_port`	int	`0x10`	`0x0` .. `0xFE`	Serial (6850) base -- SW-2. 0x10 = 2SIO Port A (default); SW-2 ON = 0x18
`standard`	enum	`mits`	`mits` | `kansas`	SW-1 tape modulation: mits (2400/1850) | kansas (Kansas City 2400/1200)
`motor`	enum	—	—	Tape-recorder motor relay (guest-driven: OUT 6,127 = on, OUT 6,191 = off) **(read-only — not a key you may set)**

Unit `tape` — `[board.unit.tape]`

Key	Kind	Default	Legal	Meaning
`mode`	enum	`play`	`play` | `record`	Which way the bytes go: play loads from the file, record saves to it
`format`	enum	`auto`	`auto` | `raw` | `fsk300`	How to read the mounted file: auto | raw | fsk300
`leader`	int	`15`	`0` .. `120`	Seconds of idle tone before recorded data, when writing audio
`trailer`	int	`5`	`0` .. `120`	Seconds of idle tone after recorded data, when writing audio
`waveform`	enum	`square`	`square` | `sine`	Carrier shape when writing audio: square (like real hardware) | sine
`level`	int	`36`	`1` .. `100`	Recording level as a percent of full scale, when writing audio
`rate`	enum	`full`	`full` | `real`	Playback speed: full (as fast as the guest reads) | real (wall-clock baud)
`detected`	string	—	—	What the mounted tape turned out to be (empty if nothing is mounted) **(read-only — not a key you may set)**
`position`	string	—	—	Where the tape head is now: mm:ss / total (percent) -- read-only **(read-only — not a key you may set)**
`counter`	enum	`on`	`on` | `off`	Live tape counter on the console during a load: on | off
`stop`	string	`off`	off | end | mm:ss	Auto-stop playback at this time: off | end | mm:ss

Unit `serial` — `[board.unit.serial]`

Key	Kind	Default	Legal	Meaning
`baud`	int	`9600`	`50` .. `76800`	Line rate. A JUMPER on the real card -- software cannot change it, and there is no free-running setting: the rate paces the line
`interrupt`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where this channel's IRQ is jumpered: none | int | vi0..vi7 *(interrupt strap)*
`dcd`	enum	`ground`	`ground` | `wired`	/DCD pin: grounded on the card, or wired to the connector
`cts`	enum	`ground`	`ground` | `wired`	/CTS pin: grounded on the card, or wired -- and then it gates the transmitter
`lines`	string	—	—	Live pin state (read-only). CAPITALS = asserted. in: DCD CTS, out: RTS BRK **(read-only — not a key you may set)**
`connect`	string	`null`	text	The endpoint on the other end of the line (CONNECT sets this)

Parallel and printer

`4pio`

MITS 88-4PIO: up to four 6820 PIAs, sections ja/jb.. per port. 16 ports from BASE (default 20). Software-set direction; CONNECT each section

Units: `ja` (serial, CONNECT), `jb` (serial, CONNECT)

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0x20`	`0x0` .. `0xF0`	Base address -- must be on a 16-address boundary. 16 ports from here
`ports`	int	`1`	`1` .. `4`	How many 6820 PIAs are populated (1..4 -- J, K, L, M)

Unit `ja` — `[board.unit.ja]`

Key	Kind	Default	Legal	Meaning
`connect`	string	`null`	text	The endpoint on the other end of this section (CONNECT sets this)

Unit `jb` — `[board.unit.jb]`

Key	Kind	Default	Legal	Meaning
`connect`	string	`null`	text	The endpoint on the other end of this section (CONNECT sets this)

`c700`

MITS 88-C700: Centronics line-printer controller, unit 'prn'. Two ports at BASE+0..1 (default 02). Output-only; CONNECT it to a file, a socket, or a real printer queue

Units: `prn` (serial, CONNECT)

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0x2`	`0x0` .. `0xFE`	Base address -- MUST BE EVEN. Control/status at BASE, data at BASE+1
`connect`	string	`null`	text	The endpoint on the other end of the line (CONNECT sets this)
`interrupt`	enum	`int`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where the printer's interrupt is soldered: none | int (pin 73) | vi0..vi7 *(interrupt strap)*
`interrupt_after`	enum	`char`	`char` | `crlf`	SW2 #4: raise the interrupt after every 'char' or only after a 'crlf'

`d7a`

Cromemco D+7A: analog + parallel I/O. Eight ports from BASE (default 18): one parallel port + seven two's-complement A/D-in/D/A-out channels. Reads 1-2 JS-1 joysticks from the host

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0x18`	`0x0` .. `0xF8`	Base of the 8-port block (A7..A3 jumpers): parallel at BASE, analog at BASE+1..7. A multiple of 8; default 18
`joystick1`	string	`auto`	text	Which host controller drives JS-1 console 1: 'none', 'auto' (the matching gamepad -- console 1->pad 0, console 2->pad 1 -- or the keyboard), 'keyboard', or a device index like 0
`joystick2`	string	`auto`	text	Which host controller drives JS-1 console 2: 'none', 'auto' (the matching gamepad -- console 1->pad 0, console 2->pad 1 -- or the keyboard), 'keyboard', or a device index like 0
`speaker1`	string	`1`	text	The analog output that JS-1 console 1's speaker is on: 'none', or a channel 1 to 7 (channel n is port BASE+n). Default 1
`speaker2`	string	`3`	text	The analog output that JS-1 console 2's speaker is on: 'none', or a channel 1 to 7 (channel n is port BASE+n). Default 3

`lpc`

MITS 88-LPC: 88-LP line-printer controller, unit 'prn'. Two ports at BASE+0..1 (default 02). Line-buffered: 6-bit codes + PRINT/LINE FEED/CLEAR. CONNECT it to a file, a socket, or a real printer queue

Units: `prn` (serial, CONNECT)

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0x2`	`0x0` .. `0xFE`	Base address -- MUST BE EVEN. Control/status at BASE, data at BASE+1
`connect`	string	`null`	text	The endpoint on the other end of the line (CONNECT sets this)

`music6`

Newtech Model 6 Music Board: one write-only port into a 6-bit D/A, an amplifier and a speaker. Answers at four addresses from BASE (default 24); DO7..DO2 are latched. A program makes the sound by writing the port in a timed loop. Needs a crystal (clock_hz) to play

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0x24`	`0x0` .. `0xF4`	The output port (jumpers J1..J8 set A7..A4): 04, 14, 24 ... F4. The board answers at this address and the next three. Default 24

`pio`

MITS 88-PIO: 8-bit parallel port, units 'out'/'in'. Two ports at BASE+0..1 (default 04). CONNECT a printer, a keyboard, a socket

Units: `out` (serial, CONNECT), `in` (serial, CONNECT)

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0x4`	`0x0` .. `0xFE`	Base address -- MUST BE EVEN. Control/status at BASE, data at BASE+1

Unit `out` — `[board.unit.out]`

Key	Kind	Default	Legal	Meaning
`connect`	string	`null`	text	The endpoint on the other end of this line (CONNECT sets this)

Unit `in` — `[board.unit.in]`

Key	Kind	Default	Legal	Meaning
`connect`	string	`null`	text	The endpoint on the other end of this line (CONNECT sets this)

Video

`cadzilla`

CADzilla: an HD63484 ACRTC graphics board with a Bt453 RAMDAC and 2 MB of fixed frame memory, on a fixed VESA monitor (mode: 640x480, 800x600, 1024x768 (default)). One 8-port I/O block at BASE (default 70): ACRTC RS=0 at +0, MODE register at +1 (write-only: HSPOL/VSPOL/AMODE/OLEN), ACRTC RS=1 at +2, Bt453 at +4..+7. Draws by command through the ACRTC FIFO; wired for 8 bpp, GAI +8, single or interleaved access set by MODE AMODE. Interrupts (SW1-8) optional (interrupt=none|int|vi0..vi7). Needs a Display

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0x70`	`0x0` .. `0xF8`	I/O base -- one 8-port block: BASE the ACRTC address/status, BASE+1 the MODE register, BASE+2 the ACRTC data/FIFO port, BASE+4..+7 the Bt453. A multiple of 8; default 70
`mode`	enum	`1024x768`	`640x480` | `800x600` | `1024x768`	The monitor: a fixed-frequency VESA raster the ACRTC's picture is placed in by its HDS/VDS. 640x480, 800x600 or 1024x768 (default)
`draw_rate`	enum	`full`	`full` | `real`	Drawing speed. full: each ACRTC command completes at once. real: each command takes its HD63484 data-sheet time, and the write FIFO fills as on the real board
`width`	string	`auto`	text	Video window width in pixels: 'auto' (default) opens about half the screen wide, or a number like 1024. The height follows the board's own aspect, and the picture is a whole multiple of its pixels so it stays crisp
`interrupt`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	SW1-8: the S-100 line for the ACRTC's IRQ*. none (default) disconnects it. int (pin 73) or vi0..vi7 is asserted while an enabled status flag is set *(interrupt strap)*
`video`	string	—	—	LIVE: whether the ACRTC is displaying -- OMR STR and DCR SE1 both set. Read-only **(read-only — not a key you may set)**
`picture`	string	—	—	LIVE: the size in pixels of the picture that the ACRTC is set to show, and the position of its top-left corner in the monitor's frame. Read-only **(read-only — not a key you may set)**
`wiring`	string	—	—	LIVE: 'ok' when the ACRTC settings agree with the board's wiring (CCR GBM 8 bpp, OMR GAI +8 words, OMR ACM the same as MODE AMODE, no displayed screen with MWR CHR set). If not, it names the setting that is wrong, and the picture is scrambled. Read-only **(read-only — not a key you may set)**
`hspol`	string	—	—	LIVE: MODE register bit 0, the horizontal sync polarity. The simulator shows the value but does not use it. Read-only **(read-only — not a key you may set)**
`vspol`	string	—	—	LIVE: MODE register bit 1, the vertical sync polarity. The simulator shows the value but does not use it. Read-only **(read-only — not a key you may set)**
`amode`	string	—	—	LIVE: MODE register bit 2, the access mode of the board's fetch logic: single or interleaved. The ACRTC's OMR ACM bit must agree (see wiring). Read-only **(read-only — not a key you may set)**

`olen`	bool	—	—	LIVE: MODE register bit 3, overlay display enable. Off, the overlay is not shown; drawing into it still works. Read-only **(read-only — not a key you may set)**
`olsel`	int	—	—	LIVE: MODE register bit 4, overlay select. Which half of the overlay memory a CHR screen draws into: 0 the half for the lower 1 MB of frame memory, 1 the upper. Read-only **(read-only — not a key you may set)**
`status`	int	—	—	LIVE: the ACRTC status register -- CER ARD CED LPD RFF RFR WFR WFE. Read-only **(read-only — not a key you may set)**
`irq`	bool	—	—	LIVE: whether IRQ* is asserted right now -- an enabled status flag pending AND the interrupt strap not 'none'. Read-only **(read-only — not a key you may set)**

`dazzler`

Cromemco Dazzler: color graphics from a framebuffer in main RAM. Two ports at BASE+0..1 (default 0E): control/status and format. 32x32 to 128x128, 16 colors/greys. Needs a Display

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0xE`	`0x0` .. `0xFE`	I/O base port -- control/status (BASE) and format (BASE+1). Even; default 0E
`width`	string	`auto`	text	Video window width in pixels: 'auto' (default) opens about half the screen wide, or a number like 1024. The height follows the board's own aspect, and the picture is a whole multiple of its pixels so it stays crisp
`video`	string	—	—	LIVE: whether the Dazzler is displaying -- OUT BASE D7 (on/off). Read-only **(read-only — not a key you may set)**
`resolution`	string	—	—	LIVE: picture size in elements, decoded from the format byte (D6 X4, D5 size): 32x32, 64x64 or 128x128. Read-only **(read-only — not a key you may set)**
`color`	string	—	—	LIVE: color vs black-and-white -- format D4. Read-only **(read-only — not a key you may set)**
`size`	string	—	—	LIVE: framebuffer footprint -- format D5: 512 bytes (one quadrant) or 2 KB (four quadrants). Read-only **(read-only — not a key you may set)**
`base`	int	—	—	LIVE: framebuffer start address in RAM -- OUT BASE D6-D0 << 9. Read-only **(read-only — not a key you may set)**

`dazzler2`

Cromemco Dazzler II (a modern board): the Dazzler with its picture in 4 KB of RAM on the card, not main RAM. It watches memory writes, so the picture can sit in a ROM or an empty address, and it holds undefined data until a base is set. Ports 0E/0F as the Dazzler. Needs a Display

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0xE`	`0x0` .. `0xFE`	I/O base port -- control/status (BASE) and format (BASE+1). Even; default 0E
`width`	string	`auto`	text	Video window width in pixels: 'auto' (default) opens about half the screen wide, or a number like 1024. The height follows the board's own aspect, and the picture is a whole multiple of its pixels so it stays crisp
`video`	string	—	—	LIVE: whether the Dazzler is displaying -- OUT BASE D7 (on/off). Read-only **(read-only — not a key you may set)**
`resolution`	string	—	—	LIVE: picture size in elements, decoded from the format byte (D6 X4, D5 size): 32x32, 64x64 or 128x128. Read-only **(read-only — not a key you may set)**
`color`	string	—	—	LIVE: color vs black-and-white -- format D4. Read-only **(read-only — not a key you may set)**
`size`	string	—	—	LIVE: framebuffer footprint -- format D5: 512 bytes (one quadrant) or 2 KB (four quadrants). Read-only **(read-only — not a key you may set)**
`base`	int	—	—	LIVE: framebuffer start address in RAM -- OUT BASE D6-D0 << 9. Read-only **(read-only — not a key you may set)**
`page`	enum	`0`	`0` | `800`	Which 2 KB half of the card's 4 KB RAM is shown: 0 = the first, 800 = the second, at +800H (jumper P18 pins 31-32 closed)
`seed`	int	`1`	any	Seed for the undefined contents of the card RAM at power-on. The same seed gives the same contents at every power

`vdb8024`

SD Systems VDB-8024: an 80x24 video terminal on one board -- the video console for an SBC-100/200 (the alternative to the 8251). Two I/O ports at BASE+0..1 (default 00): status/keyboard/display. Unit 'keyboard' (CONNECT). Optional keyboard-strobe interrupt strap (interrupt=vi0..vi7) for the SBC-200's CTC to vector -- what the SD video CBIOS needs; polled by default. Boots sdmonv21. Needs a Display

Units: `keyboard` (serial, CONNECT)

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0x0`	`0x0` .. `0xFE`	Low I/O port: status (IN base+0) / keyboard (IN base+1) / display (OUT base+1). The real card is fixed at 00
`cursor`	enum	`blink`	`off` | `blink` | `steady`	Cursor at the current cell: off, blink, or steady (the board default is a blinking cursor)
`video`	enum	`normal`	`normal` | `reverse`	Screen video polarity: normal (light on dark) or reverse
`interrupt`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Keyboard-strobe interrupt strap: none = polled (default), or the S-100 VI line the keyboard raises while a byte waits (the SBC-200 CTC vectors it -- video CBIOS straps vi2) *(interrupt strap)*
`width`	string	`auto`	text	Video window width in pixels: 'auto' (default) opens about half the screen wide, or a number like 1024. The height follows the board's own aspect, and the picture is a whole multiple of its pixels so it stays crisp

Unit `keyboard` — `[board.unit.keyboard]`

Key	Kind	Default	Legal	Meaning
`connect`	string	`null`	text	The endpoint on the other end of the keyboard line (CONNECT sets this)

`vdm1`

Processor Technology VDM-1: memory-mapped 16x64 video, screen RAM at BASE (default CC00), scroll/status port (default CC). MCM6576 character ROM; control codes 00-1F show their glyphs unless `blanking` says not. Needs a Display

Board properties

Key	Kind	Default	Legal	Meaning
`base`	int	`0xCC00`	`0x0` .. `0xFC00`	Screen-RAM base address -- 1 KB-aligned (16x64 = 1024 bytes)
`port`	int	`0xCC`	`0x0` .. `0xFC`	I/O port -- scroll (OUT) / status (IN). Low two bits are zero
`video`	enum	`normal`	`normal` | `reverse`	Video polarity (SW1/SW2): normal (light on dark) or reverse
`cursor`	enum	`blink`	`off` | `blink` | `steady`	Cursor for a byte with bit 7 set (SW3/SW4): off, blink, or steady
`blanking`	enum	`none`	`none` | `crvt` | `control` | `all`	Control-code and text blanking (SW5/SW6): none = every code shows its glyph; crvt = also a CR blanks the rest of its line and a VT the rest of the screen; control = crvt, and codes 00-1F are blank; all = only cursor blocks show
`fill`	enum	`random`	`zero` | `random`	Screen RAM at power-on: zero | random (real RAM is not zeroed; 0x00 shows as a box)
`seed`	int	`1`	any	Seed for fill=random. The same seed fills the screen the same way at every POWER, so a run is repeatable; change it for a different pattern
`width`	string	`auto`	text	Video window width in pixels: 'auto' (default) opens about half the screen wide, or a number like 1024. The height follows the board's own aspect, and the picture is a whole multiple of its pixels so it stays crisp

Systems

`sol`

Processor Technology Sol-PC I/O: serial, keyboard, parallel, CUTS tape as one board. Seven ports F8..FE. Units serial/printer/keyboard (CONNECT) and tape1/tape2 (MOUNT). Brings the WIND/REWIND/EXTRACT verbs and a tape counter

Units: `serial` (serial, CONNECT), `printer` (serial, CONNECT), `keyboard` (serial, CONNECT), `tape1` (tape, MOUNT), `tape2` (tape, MOUNT)

Board properties

Key	Kind	Default	Legal	Meaning
`base`	int	`0xF8`	`0x0` .. `0xF8`	Base I/O port (decodes BASE+0..6). Fixed at F8 on a real Sol-PC

Unit `serial` — `[board.unit.serial]`

Key	Kind	Default	Legal	Meaning
`connect`	string	`null`	text	The endpoint on the other end of this line (CONNECT sets this)
`baud`	int	`9600`	`1` .. `1000000`	Serial line speed (the strap on the Sol-PC's serial UART)
`data_bits`	int	`8`	`5` .. `8`	Serial word length: 8, 7, or 6 (the Sol-PC DIP)

Unit `printer` — `[board.unit.printer]`

Key	Kind	Default	Legal	Meaning
`connect`	string	`null`	text	The endpoint on the other end of this line (CONNECT sets this)

Unit `keyboard` — `[board.unit.keyboard]`

Key	Kind	Default	Legal	Meaning
`connect`	string	`null`	text	The endpoint on the other end of this line (CONNECT sets this)

Unit `tape1` — `[board.unit.tape1]`

Key	Kind	Default	Legal	Meaning
`mode`	enum	`play`	`play` | `record`	Which way the bytes go: play loads from the file, record saves to it
`format`	enum	`auto`	`auto` | `raw` | `cuts1200` | `kcs300`	How to read the mounted file: auto | raw | cuts1200 | kcs300
`leader`	int	`3`	`0` .. `120`	Seconds of idle tone before recorded data, when writing audio
`trailer`	int	`2`	`0` .. `120`	Seconds of idle tone after recorded data, when writing audio
`waveform`	enum	`square`	`square` | `sine`	Carrier shape when writing audio: square (like real hardware) | sine
`level`	int	`36`	`1` .. `100`	Recording level as a percent of full scale, when writing audio
`rc`	int	`4000`	`1000` .. `20000`	Edge-rounding low-pass corner in Hz, when writing CUTS audio
`rate`	enum	`full`	`full` | `real`	Playback speed: full (as fast as the guest reads) | real (wall-clock baud)
`detected`	string	—	—	What the cassette in this deck turned out to be (empty if none) **(read-only — not a key you may set)**
`position`	string	—	—	Where this deck's head is now: mm:ss / total (percent) -- read-only **(read-only — not a key you may set)**
`counter`	enum	`on`	`on` | `off`	Live tape counter on the console during a load: on | off
`stop`	string	`off`	text	Auto-stop playback at this time: off | end | mm:ss

Unit `tape2` — `[board.unit.tape2]`

Key	Kind	Default	Legal	Meaning
`mode`	enum	`play`	`play` | `record`	Which way the bytes go: play loads from the file, record saves to it
`format`	enum	`auto`	`auto` | `raw` | `cuts1200` | `kcs300`	How to read the mounted file: auto | raw | cuts1200 | kcs300
`leader`	int	`3`	`0` .. `120`	Seconds of idle tone before recorded data, when writing audio
`trailer`	int	`2`	`0` .. `120`	Seconds of idle tone after recorded data, when writing audio
`waveform`	enum	`square`	`square` | `sine`	Carrier shape when writing audio: square (like real hardware) | sine
`level`	int	`36`	`1` .. `100`	Recording level as a percent of full scale, when writing audio
`rc`	int	`4000`	`1000` .. `20000`	Edge-rounding low-pass corner in Hz, when writing CUTS audio
`rate`	enum	`full`	`full` | `real`	Playback speed: full (as fast as the guest reads) | real (wall-clock baud)
`detected`	string	—	—	What the cassette in this deck turned out to be (empty if none) **(read-only — not a key you may set)**
`position`	string	—	—	Where this deck's head is now: mm:ss / total (percent) -- read-only **(read-only — not a key you may set)**
`counter`	enum	`on`	`on` | `off`	Live tape counter on the console during a load: on | off
`stop`	string	`off`	text	Auto-stop playback at this time: off | end | mm:ss

PROM programmer

`pb1`

SSM PB1: 2708/2716 EPROM programmer + on-board EPROM board. A 4K programming-socket window (default D000, sockets U22=2708/U23=2716) and one control port (default 10): OUT arms the board and picks the chip (D0=2708, D1=2716), then a window write burns a byte and a window read disarms it. Save the burn to a host hex file with `SAVE file window`. Optional read-only on-board EPROM area via [[board.prom]] (at + mount). The board ships no firmware -- run any 2708/2716 burner, such as SSM's own from the PB1 manual

Units: `u22` (rom, MOUNT), `u23` (rom, MOUNT)

`[[board.socket]]` — a list you may add

Key	Kind	Legal	Meaning
`chip`	enum	`2708` | `2716`	Which programming socket: 2708 (U22, 1K) or 2716 (U23, 2K)
`mount`	string	text	A source or erased chip image to put in the socket. Relative to THIS FILE.

`[[board.prom]]` — a list you may add

Key	Kind	Legal	Meaning
`at`	int	`0x8000` .. `0xFFFF`	Where the on-board EPROM sits. An address at or above 8000.
`mount`	string	text	The firmware image: a file (relative to THIS FILE), or builtin:.

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0x10`	`0x0` .. `0xF0`	Control port: an OUT here arms the board and picks 2708 (D0) or 2716 (D1). Only A4-A7 decode, so it must be an x0H address (00, 10, .. F0)
`window`	int	`0xD000`	`0x0` .. `0xF000`	The 4K programming-socket window, on a 4K boundary (0000, 1000, .. F000). The guest writes bytes here to burn, and reads here to disarm

Other

`fp`

Altair front panel: the SENSE switches a guest reads at IN 0FFH -- a configured byte (SET fp0 sense= or TOML), not toggled here. No OUT

Board properties

Key	Kind	Default	Legal	Meaning
`sense`	int	`0x0`	`0x0` .. `0xFF`	The SENSE switches, SA8..SA15 -- what IN 0FFH reads

`hostbridge`

Host Bridge: guest <-> host file transfer, sandboxed. OUR OWN BOARD, not a period one. Two ports at BASE+0..1. R.COM/W.COM/HDIR.COM

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0xB0`	`0x0` .. `0xFE`	Base port. Two ports: BASE+0 command/status, BASE+1 data
`hostdir`	string		text	The sandbox root. Guest names resolve here and CANNOT escape it. Empty = the shell's working directory
`hostdir_root`	string	—	—	LIVE: the sandbox root as RESOLVED -- the actual directory the guest is fenced into. Read-only; `hostdir` is what was written. **(read-only — not a key you may set)**
`readonly`	bool	`false`	`on` | `off`	Refuse OPEN_WRITE and DELETE -- the guest may read the host, not change it

`rtc100`

SciTronics RTC-100: an S-100 battery-backed real-time clock/calendar (OKI MSM5832) behind a 6821 PIA. Four consecutive ports from a base that must be a multiple of 4 (port A data/direction at base+0 -- digit address in the low nibble, digit data in the HIGH nibble on a read; port A control at base+1 -- CA2 is the clock's Hold, low = stopped; port B at base+2 -- bit 0 is the Write strobe; port B control at base+3 -- CB2 is the Read line). Keeps time from the host, battery-backed across RESET, and settable by the guest. Optional once-a-second interrupt on pin 73, vectored by the card itself with RST 0-7 (the `restart` switch)

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0xF0`	`0x0` .. `0xFC`	Base address -- MUST BE A MULTIPLE OF 4. Four consecutive ports
`interrupt`	enum	`none`	`none` | `int`	The once-a-second interrupt: none | int (pin 73, vectored by `restart`)
`restart`	int	`7`	`0` .. `7`	INT switch: which RST the card jams on acknowledge, 0-7 (vector 8n). 0 and 7 are legal but commonly taken by other devices
`time`	string	—	—	LIVE: the date/time the MSM5832 is showing, and its offset from host time **(read-only — not a key you may set)**

`ss1`

CompuPro System Support 1: multifunction S-100 board. Dual 8259A interrupt controllers in a master/slave cascade (master/slave at base+0..+3; master watches VI0-6 and drives pin 73, slave takes

the timer OUTs and the UART's Rx/TxRDY), an 8253 interval timer (three counters + control at base+4..+7, 2 MHz clock), the OKI MSM5832 battery-backed real-time clock/calendar (command/data at base+10/+11) and a 2651 UART serial channel (base+12..+15); base default 50H. The math socket (base+8/+9) takes an AMD 9511A or 9512 with `math`, and is empty by default

Units: `serial` (serial, CONNECT)

Board properties

Key	Kind	Default	Legal	Meaning
`base`	int	`0x50`	`0x0` .. `0xF0`	Base port of the 16-port I/O block. CompuPro standard is 50H
`math`	enum	`none`	`none` | `9511a` | `9512`	The chip in the math socket (U13): none (the stock empty socket), 9511a (AMD Am9511A, Intel 8231A) or 9512 (AMD Am9512, Intel 8232)
`clock`	string	—	—	LIVE: the date/time the MSM5832 is showing, and its offset from host time **(read-only — not a key you may set)**
`timer`	string	—	—	LIVE: the three 8253 counters -- mode, current count and OUT level **(read-only — not a key you may set)**
`pic`	string	—	—	LIVE: the master/slave 8259A -- IRR/ISR/IMR and the master's INT pin **(read-only — not a key you may set)**
`mathchip`	string	—	—	LIVE: the math chip -- busy/idle, stack pointer, status, top of stack, END/SVREQ/ERR **(read-only — not a key you may set)**

Unit `serial` — `[board.unit.serial]`

Key	Kind	Default	Legal	Meaning
`baud`	int	`9600`	`50` .. `19200`	Line rate. The 2651 generates it from Mode Register 2, so the guest's MR2 write overwrites this; it seeds the line before then (min 50)
`interrupt`	enum	`none`	`none` | `int` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where this port's IRQ is jumpered: none | int | vi0..vi7. The chip raises it on RxRDY; the standard board routes it through the 8259A instead *(interrupt strap)*
`connect`	string	`null`	text	The endpoint on the other end of the line (CONNECT sets this)

`virtc`

MITS 88-VI/RTC: vectored interrupts (VI0-VI7 -> RST n) and a real-time clock. One port at FE

Board properties

Key	Kind	Default	Legal	Meaning
`port`	int	`0xFE`	`0x0` .. `0xFF`	Control port. 0xFE (376 octal) on the real card -- write only
`rtc_source`	enum	`line`	`line` | `clock`	RTC clock source jumper: the 60 Hz line, or 10 kHz off the 2 MHz clock
`rtc_divide`	enum	`1`	`1` | `10` | `100` | `1000`	RTC divider jumper: source frequency / 1, 10, 100 or 1000
`rtc_interrupt`	enum	`none`	`none` | `vi0` | `vi1` | `vi2` | `vi3` | `vi4` | `vi5` | `vi6` | `vi7`	Where the RTC's interrupt ("RI") is jumpered: none | vi0..vi7. Leave it `none` to run the PS2 package *(interrupt strap)*
`vi_enabled`	bool	—	—	LIVE: is the 88-VI structure enabled? (control bit 7; POC clears it) **(read-only — not a key you may set)**
`level_live`	bool	—	—	LIVE: is the current-level comparison in circuit? (control bit 3) **(read-only — not a key you may set)**
`rtc_pending`	bool	—	—	LIVE: has the RTC's interrupt flip-flop set? (cleared by control bit 4) **(read-only — not a key you may set)**
`current_level`	int	—	—	LIVE: the current interrupt level (control bits 0-2, ones-complement on the wire). Nothing at this level or below may interrupt while level_live **(read-only — not a key you may set)**

The built-in machines

A built-in is a machine file that lives **inside the binary** — the same format you would write yourself, with nothing special about it. Name one and you get it in any directory on earth:

```
$ altairsim basic4k
```

`altairsim --list` prints this table, and `altairsim -x 'SHOW MACHINE' <name>` shows what is actually in

one.

Machine	What it is
`8085`	A minimal 8085 machine: an `8085` CPU, 64K of RAM, and a 2SIO console.
`acuter`	ACUTER at F000 -- CUTER on a plain Altair, with a terminal instead of a VDM.
`altmon`	An Altair with a monitor in ROM and a terminal on it.
`amon`	AMON 3.1 in a 4K EPROM at F000 -- Martin Eberhard's full-featured Altair monitor.
`bankmem`	A bank-switched RAM machine: a Z80, a console, and a Vector Graphic 64K bankmem.
`basic4k`	The machine Altair 4K BASIC was sold to run on: an 88-SIO Teletype, a cassette in the ACR.
`basic8k`	The machine Altair 8K BASIC was sold to run on: an 88-2SIO terminal, a cassette in the ACR.
`cadzilla`	An HD63484 ACRTC graphics board with a Bt453 RAMDAC in an Altair -- the bench for CADzilla.
`cdbl`	The `default` machine with the Combo Disk Boot Loader in the PROM socket.
`compupro`	A stock Altair with a CompuPro System Support 1 board for its clock/calendar.
`cuter`	CUTER 1.3 driving a Processor Technology VDM-1 -- the real Sol/CUTS monitor.
`dazzler`	A Cromemco Dazzler in an Altair -- the bench for the S-100's first color graphics card.
`default`	The machine you get when you name none: 56K, and the DBL boot PROM at FF00.
`disk1`	CompuPro Disk 1 floppy: the controller at C0, its boot EPROM and an Interfacer I console. You supply the disk.
`disk1a`	CompuPro Disk 1A floppy: the controller at C0, its boot EPROM and a System Support 1 console. You supply the disk.
`dualide`	S100Computers "IDE-AB CF" machine -- a V2 Z80 CPU board booting CP/M 3 off a CompactFlash card.
`dualidesd`	S100Computers "IDE-AB CF+ESP32" machine -- both boards, CP/M 3 on CF drives A:/B: and SD drives C:/D:.
`dualsd`	S100Computers "Dual SD" machine -- a V2 Z80 CPU board booting CP/M 3 off a microSD card.
`icom`	iCOM FD3712 8" floppy machine -- boots CP/M 2.2 off a single-density iCOM disk.
`lineprinter-lpc`	The `default` machine with an 88-LPC line printer at port 02; CONNECT `lpt0:prn` to a file or the console.
`lineprinter`	The `default` machine with an 88-C700 line printer at port 02; CONNECT `lpt0:prn` to a file or the console.
`minidisk`	The Altair Minidisk: an 88-MDS at 08 and the MDBL boot PROM. You supply the 5.25" disk.
`northstar`	North Star MDS-A single-density floppy: the controller at E800 and its boot PROM. You supply the disk.
`northstardd`	North Star MDS-A-D double-density floppy: the controller at E800 and its boot PROM. You supply the disk.

`original`	The Altair as it actually left Albuquerque.
`parallel`	The `default` machine with two MITS parallel boards: an 88-PIO and an 88-4PIO.
`ps2`	The machine MITS Programming System II ran on: basic8k's cards, but not basic8k's bootstrap.
`ps2int`	MITS Programming System II, WITH INTERRUPTS. `ps2` with A9 down and an 88-VI/RTC in it.
`rombasic`	MITS Extended ROM BASIC 16K -- Extended BASIC that runs directly out of ROM.
`sbc200`	SD Systems SBC-200 -- a 4 MHz Z80 single-board computer running the MSMONR21 monitor.
`sbc200v`	The SD Systems SBC-200 with a VDB-8024 VIDEO console: SDMONV21 on an 80x24 screen.
`sol20`	The Processor Technology Sol-20 -- an integrated 8080 machine, running SOLOS.
`tarbell`	Tarbell #1011 single-density floppy machine -- boots CP/M 2.2 the moment a disk is in it.
`tarbelldd`	Tarbell #2022 double-density floppy machine -- boots CP/M 2.2 the moment a disk is in it.
`turnkey`	The MITS 8800bt -- an Altair with a Turnkey Module where the front panel used to be.
`vdm1`	A Processor Technology VDM-1 in an Altair, and a demo that draws on it.
`z80`	A minimal Z80 machine: a `z80` CPU, 64K of RAM, and a 2SIO console.