



The Serial Console

A field guide to **serial-console-mcp** — how an agent opens a craft port, a CAT radio, a rotator or a microcontroller over a serial cable, picks the right preset, says the right line ending, reads until the prompt instead of guessing, sees a VT100 screen the way you would, keys a line when asked, and hands the port back cleanly. Everything a terminal program knows, explained for the operator who would rather ask.

FIELD GUIDE · VERSION 0.3 · SEPTEMBER 2026

30

TOOLS

12

DEVICE
PRESETS

6

CONSOLE
RULES

5

WORKED
SESSIONS

Contents



BEFORE YOU START

A	The tools at a glance	03
B	Installing	05
C	Your first session — a console by conversation	06
D	How the system works	07

THE SETTINGS AND THE RULES

01	Connection settings — the Quick Connect dialog, in words	08
02	The console standard	10
03	The device playbook	11
04	The shack — several ports, control lines, scripts, capture	13
05	Terminals — clean text, keys, and a real screen	15

AT THE BENCH

E	Worked sessions	17
F	When it doesn't answer	20

REFERENCE

G	Tool reference	22
H	About & provenance	24

The Tools at a Glance

One server, thirty tools, as many open ports as the shack has cables. The agent picks the tool; you say what you want in plain language.

TOOLS	WHAT THEY DO	WHEN THE AGENT REACHES FOR THEM
<code>list_serial_ports</code> · <code>list_presets</code>	Every port the computer sees, marked if open here; the usual settings for ten device families.	First, whenever a port or its settings are unknown
<code>connect</code> · <code>reconnect_last</code> · <code>disconnect</code>	Open a port by name with baud, framing, flow control, line ending and prompt, or from a preset. Several at once. Remembered across sessions.	“Connect the Kenwood as rig and the rotator on the other port”
<code>send_text</code> · <code>send_hex</code>	Write an ASCII line or raw bytes. Writes only; never reads.	Any command, login, bare return, CI-V frame
<code>read_until_prompt</code> · <code>read_available</code> · <code>query_text</code>	Return buffered output up to a prompt (leaving the rest), or whatever has arrived, or clear-send-read in one.	Every reply
<code>expect</code>	A scripted list of send-and-wait steps in one call, with automatic replies to pagers.	Logins, multi-step command sequences
<code>send_keys</code> · <code>screen</code>	Press Ctrl-C, Esc, Tab, arrows and F-keys by name; view the VT100/xterm screen of a full-screen console.	“Interrupt that ping” · BIOS, BMC and menu consoles
<code>set_lines</code> · <code>pulse_line</code> · <code>send_break</code>	Drive DTR and RTS, send BREAK; status shows CTS/DSR/CD/RI.	Key a rig's PTT, reset an Arduino, interrupt a boot
<code>capture_start</code> · <code>capture_stop</code> · <code>get_transcript</code>	Log a session to a file, raw or timestamped; re-read the rolling transcript.	“Record this console session”
<code>port_in_use_by</code> · <code>detect_baud</code>	Which program holds a port; which baud rate produces readable text.	“It says in use” · “I don't know the rate”
<code>cat_build</code> · <code>cat_parse</code>	Kenwood/Elecraft/Yaesu ';' commands: build a frequency set, decode ID, FA, MD, IF and error replies.	Any text-CAT rig
<code>civ_build</code> · <code>civ_parse</code> · <code>civ_freq</code>	Icom CI-V frames: build, decode (echo vs reply, BCD frequency, mode, PTT), convert.	Any Icom
<code>rotator_build</code> · <code>rotator_parse</code>	GS-232 commands (read, move, stop, speed) and position replies.	Any Yaesu-protocol rotator
<code>clear_buffer</code> · <code>status</code>	Housekeeping; status shows every open port, its lines, buffer and capture.	Pre-flight; after anything odd

WHO READS THIS GUIDE

The agent reads the tool descriptions. This guide is for the human at the bench: what the agent will do on your behalf, which words make it do the right thing, and how to tell when a silent port is a wrong baud rate rather than a dead device.

Installing

The server runs as a host process next to Claude Desktop, not inside its sandbox. That is the whole reason it can see `/dev/cu.*`, `COM4` or `/dev/ttyUSB0`.

With the installer **recommended**

1 Run the installer.

Download `SerialConsoleMCP-Setup.exe` (Windows) or `SerialConsoleMCP-0.1.1.pkg` (Mac) from the GitHub Releases page and click through it. It copies one binary and registers it in `claude_desktop_config.json`, merging with any servers already there and backing up the old file. The installers are unsigned, so expect a Gatekeeper or SmartScreen warning.

2 Fully quit Claude Desktop, then reopen it.

Closing the window is not enough. Windows: right-click the tray icon, Quit. Mac: `⌘Q`. Servers are read at launch.

From PyPI **uv or pipx**

```
uvx serial-console-mcp configure --command uvx --arg serial-console-mcp
# or
pipx install serial-console-mcp
serial-console-mcp configure --command "$(which serial-console-mcp)"
```

Needs Python 3.10 or newer. `configure` writes the Claude Desktop entry, merging with servers already there and backing up the old file; `serial-console-mcp configure --remove` undoes it. Any MCP client can launch the same command over stdio. Developers: `git clone`, `uv sync`, `uv run pytest`.

Verify **thirty seconds**

1 Ask for the ports.

“What serial ports do you see?” A list with USB descriptions means the server is running and the OS sees your adapter. “No serial ports found” with the device plugged in almost always means a charge-only USB cable or a missing driver (FTDI, CP210x, CH340).

2 Open and close a port.

“Connect to the first one, show me the status, then disconnect.” Status should say the reader is running with zero bytes buffered.

ONE PROGRAM PER PORT

A serial port can be open in exactly one program. If a terminal, WSJT-X, a contest logger, the Arduino Serial Monitor or the device's own software holds it, the connect fails with “already in use”. Say *“disconnect”* before you hand the port to something else.

Your First Session — a Console by Conversation

You do not need to know a terminal program. If you can plug in a cable and describe your gear, you can operate it.

What you're actually installing

Claude is an assistant made by Anthropic, running in the Claude Desktop app. **serial-console-mcp** is best thought of as a rig-interface cable with a very patient operator on the other end: Claude types what you ask into the port, watches what comes back, and reports it. Nothing about your device changes; it sees an ordinary terminal.

Operating is a conversation

YOU TYPE	WHAT HAPPENS
What serial ports do you see?	Every port with its USB description, so you can pick the CP210x or FTDI that is your device.
Connect to /dev/cu.usbserial-10 at 9600.	Opens the port 9600 8N1 with no flow control and starts the background reader.
Connect to /dev/cu.BLTH at 38400 with XON/XOFF.	Same, with software flow control on. Every field of a terminal's Quick Connect dialog can be said this way (chapter 01).
Send a return and read until the login prompt.	Writes a bare CR, then reads until login: appears.
Log in as admin and run show version . Show me all of it.	Sends the username, waits for the password prompt, sends the password, waits for the shell prompt, runs the command, reads until the prompt again. Long output comes back whole.
Reconnect to the same port as last time.	The last port and settings are remembered across sessions.
Disconnect.	Closes the port so a terminal program or WSJT-X can have it.

YOU ARE STILL THE OPERATOR

These tools send exactly what you ask, to whatever is on the cable. Claude Desktop asks you to approve each tool call, so you see every command before it runs. Read it. A console session on a router or a rig can reconfigure, reboot, or transmit, and this server does not try to guess which commands are dangerous. If something looks wrong, decline it, and keep a real terminal handy for anything you would not want an assistant to type.

FIRST TIME? SOMETHING HARMLESS.

Start with a read-only command: `show version` on a router, `ID;` on a Kenwood, a frequency read on an Icom. Watch one full cycle, send, prompt, output, before asking for anything that changes state.

How the System Works

A serial console is not question-and-answer. It echoes what you type, prints on its own, and can dump pages. So the server never guesses when a reply is done; it keeps reading, and you tell it what the end looks like.

1 · OPEN

connect starts a reader thread

Every byte the device sends lands in a buffer from this moment on, whether or not anyone is reading.

2 · SEND

send_text · send_hex

Write only. Command + line ending on the wire. Nothing is read here.

3 · READ

read_until_prompt · read_available

Take from the buffer until the prompt appears, or until the line goes quiet. Leftover stays buffered.

4 · CLOSE

disconnect

Stops the reader, releases the port for other programs.

- **The reader owns the port.** One thread drains the OS buffer continuously. Echo, unsolicited syslog, and a three-page `show run` all arrive intact instead of being raced by ad-hoc polling. This is the model `minicom` and `expect` use.
- **Prompt, not timer.** `read_until_prompt` returns the instant the prompt shows up and hands back everything up to and including it. A fixed delay would truncate long output or waste seconds on short output; a prompt does neither.
- **Leftover is never lost.** Bytes after the matched prompt go back to the front of the buffer, so an unsolicited line that arrived mid-read is the first thing the next read sees.
- **Timeouts return what they have.** If the prompt never shows, you still get the partial output with a note to try again or change the prompt.
- **A dead port is announced.** Unplug the adapter and the reader stops; every tool then says so and asks you to reconnect, rather than timing out in silence.
- **Memory is bounded.** The buffer holds four megabytes. A device that streams unread for hours drops its oldest bytes and `status` reports how many.

THE HABIT THAT MATTERS

Tell the agent the prompt. “The prompt is `user@r1>` with a trailing space” turns every read into a clean, complete reply. Without it the agent falls back to reading until the line goes quiet, which works for one-line CAT answers and fails for anything paged.

Connection Settings

Everything in a terminal program's Quick Connect dialog, said in a sentence. The default is **9600-8-N-1** with no flow control, which is what most consoles and much radio gear expect; name only what differs.

DIALOG FIELD	PARAMETER	DEFAULT	VALUES	HOW TO SAY IT
Port	port	required	/dev/cu.usbserial-10 COM4 · /dev/ttyUSB0	“the CP210x one” works if you listed ports first
Baud rate	baud	9600	1200 ... 115200	“at 38400”
Data bits	bytesize	8	5 · 6 · 7 · 8	“7 data bits”
Parity	parity	N	N · E · 0	“even parity”
Stop bits	stopbits	1	1 · 1.5 · 2	“two stop bits”
RTS/CTS	rtscts	off	on · off	“with hardware flow control”
XON/XOFF	xonxoff	off	on · off	“with XON/XOFF”

Shorthand works too. “38400 8N1”, “9600 7E1” and “the usual Cisco console settings” all resolve to the right parameters. The agent repeats the settings back in the connect reply, so a misheard baud rate is visible before the first command goes out.

Presets: the whole dialog in one word

`connect(preset="kenwood-cat")` loads a family's usual settings, including two the dialog never asks for: the **line ending** the device expects and the **prompt** that ends its replies. Anything you say explicitly overrides the preset. Ten presets ship: `cisco-console`, `juniper-craft`, `linux-console`, `kenwood-cat`, `elecraft-cat`, `yaesu-cat`, `icom-civ`, `xiegu-civ`, `yaesu-rotator`, `arduino`, `nmea-gps`, `screen-console`. Presets also choose the **terminal mode** (chapter 05): the console presets strip colours and escape sequences, the radio presets pass raw bytes. Ask “list the presets” to see each one's values and caveats. They are factory defaults; the device's menu wins.

Don't know the rate at all? “What baud is this thing?” runs `detect_baud`: it tries the common rates, sends a return (or a command you name, such as `ID;` for a Kenwood), and ranks the rates by how readable the reply is.

Flow control, the two lines that are usually wrong

- **Leave both off unless the manual says otherwise.** Consoles, CAT ports and microcontrollers overwhelmingly run without flow control.
- **RTS/CTS needs the wires.** Many console cables and three-wire adapters do not carry RTS and CTS. Turn it on with such a cable and every write waits for a CTS that never comes; the server gives up after two seconds and tells you exactly that.
- **XON/XOFF is for text only.** It treats the bytes 0x11 and 0x13 as flow-control characters and swallows them. Never enable it for Icom CI-V or any other binary protocol; a frequency containing those bytes would silently corrupt.

Remembered across sessions

Every successful connect writes the connection, by name, with all its settings, preset and prompt, to a small file in your user profile. “Reconnect to the rig” or “reconnect to the same port as last time” reads it back. `status` shows what is remembered when nothing is open.

LINE ENDING IS A SEND SETTING, NOT A PORT SETTING

The dialog doesn't ask because a terminal sends whatever you type. Here it matters: `send_text` appends **CR** (most radios, rotators), **LF** (Unix-style consoles, Arduino), **CRLF**, or **nothing** (Kenwood-style CAT ends in `;`). Say “*this device wants LF*” once: it is stored on the connection, as is the prompt, so every later send and read uses it without being told.

The Console Standard

Six rules the tools are built around. Each one closes a hole that a naive send-and-wait would fall into.

1 Sending never reads.

`send_text` and `send_hex` put bytes on the wire and return. A reply is a separate, explicit read. This is what lets the agent log in (send, wait for `Password:`, send, wait for the shell) without guessing delays.

2 Name the prompt with its trailing space.

The match runs over everything received, including the echo of your own command. `"# "` is safe; a bare `"#"` matches the first `#` in a banner or in `show run | include #`. A regex such as `[\w.-]+[#>] ?$` handles hostnames that change after `configure`. Set the prompt once on connect (or let the preset) and every read uses it. With no prompt anywhere, the read ends when the received text ends in `#`, `>`, `$` or `%`.

3 Clear before a command whose reply must be clean.

Syslog, interface flaps and a previous timed-out read leave bytes in the buffer. `send_text(..., clear_buffer_first=true)` or `query_text` discard them so the next read starts at your echo.

4 Text prompts for CLIs, idle reads for everything else.

Routers, switches and shells: `read_until_prompt`. Binary protocols and one-line CAT answers have no prompt: `read_available` or `query_text` with an empty prompt, which returns once the line has been quiet for a beat.

5 Baud wrong looks like garbage; line ending wrong looks like silence.

A stream of `␣` and box characters means the baud rate. An echo with no reply means the device is waiting for the other line ending. Send a bare return first to draw a fresh prompt; if that also produces nothing, check the cable.

6 Disconnect before handing the port to anything else.

The OS gives a port to one process. A forgotten open port is the most common reason WSJT-X or a terminal program “can't find” a device that is plainly plugged in.

WHERE THE RULES COME FROM

They are the reasons `minicom` and `expect` exist. The server implements them so the agent doesn't have to rediscover them at your bench.

The Device Playbook

Four families cover nearly everything on a serial cable. Settings below are the common factory defaults; the device's manual or menu wins when they differ.

FAMILY	TYPICAL SETTINGS	LINE ENDING	READ WITH	NOTES
Network craft / console ports Juniper · Cisco · Arista · switches , firewalls	9600 8N1	LF or CR	<code>read_until_prompt</code> "login: " · "> " · "# "	Send a bare return to draw a prompt. Turn off paging once (<code>set cli screen-length 0</code> , <code>terminal length 0</code>) or <code>--More--</code> becomes your prompt.
Text CAT radios Kenwood · Elecraft · Yaesu (FT-991, FTdx) · Flex	4800 – 115200 8N1 per the rig's menu	NONE	<code>query_text</code> (empty prompt, idle read)	Commands end in ; with no CR: <code>ID;</code> <code>FA;</code> <code>MD;</code> . Replies are one line ending in ; . <code>cat_build</code> formats a frequency set, <code>cat_parse</code> names the rig, mode and frequency. Elecraft echoes nothing; Kenwood may need <code>AI0;</code> to stop auto-info.
Icom CI-V IC-7300 · 7610 · 705 · 9700	19200 8N1 (USB) 9600 via CI-V jack	n/a (binary)	<code>send_hex → read_available</code>	Frames are <code>FE FE <rig> E0 <cmd> ... FD</code> ; the reply shows as hex. Frequency is BCD, low byte first. Never enable XON/XOFF. If the rig echoes, the first frame back is your own. Xiegu G90/X6100 speak CI-V too (address 0x70, <code>xiegu-civ</code> preset). Baofeng-class handhelds have no CAT: their cable carries a memory-programming protocol, which is CHIRP's job, not a console.
Rotators, switches, MCUs Yaesu GS-232 · Green Heron · Arduino · ESP32	9600 or 115200 8N1	CR (rotators) LF (Arduino)	<code>read_available</code> or <code>read_until_prompt</code>	GS-232 <code>C</code> returns <code>+0xxx</code> azimuth; <code>rotator_build</code> and <code>rotator_parse</code> cover read, move, stop and speed. Opening a port resets many Arduinos via DTR; wait a second and expect a boot banner. Sketches that print continuously are a <code>read_available</code> job.

A CAT command can key the radio (`TX;`) and a console command can reboot a router. The server doesn't filter; the approval prompt is your filter. On a rig, start with reads (`ID;` `FA;` `MD;`) and put the radio on a dummy load before any command that could transmit.

Several Ports, Control Lines, Scripts, Capture

A station is more than one cable. This chapter is what 0.2 adds on top of the console model: naming ports, driving the lines that key and reset, scripting a login in one call, and keeping a record.

Several ports at once

Every connection has a name: “connect the Kenwood as **rig**, then the rotator on the other port as **rotator**.” Tools act on the most recently used connection unless you say otherwise: “ask the rig for its frequency, then turn the rotator to 180.” `status` lists all of them and marks the current one. “Disconnect everything” closes them all. Connecting to a port that is already open replaces that connection; reusing a name for a different port is refused, so `rig` can never silently become the amplifier.

Control lines

TOOL	WHAT IT DOES	TYPICAL USE
<code>set_lines</code>	Set DTR and/or RTS and hold them	Hold an Arduino in reset; assert PTT for as long as the user says
<code>pulse_line</code>	Drive DTR or RTS to a level for N ms, then restore it, always	Reset a board (DTR low, 100 ms); key a rig for a timed transmission (RTS, 2000 ms)
<code>send_break</code>	Hold TX low for N ms	Enter ROMMON, interrupt a boot
<code>status</code>	Shows CTS, DSR, CD, RI, DTR, RTS	“Is the cable wired for flow control?”

A LINE CAN BE A TRANSMITTER

Many rig interfaces key PTT from RTS or DTR. Asking the agent to pulse a line on such a port is a transmission: the skill tells it to confirm first, keep pulses short, and never leave PTT asserted with `set_lines` unless you said so. Read-only mode refuses these tools outright.

Scripting with expect

A login is four round trips: return, username, password, first command. `expect` runs them as one call, each step a send and the prompt to wait for, and returns every step's output. A step can also just wait, or just send. `auto_reply` answers pagers on the way: `{"--More--": " "}` turns a paged `show run` into one result. If a step's prompt never shows, the sequence stops there and says so, with whatever arrived.

```
expect(steps=[
    {"send": "", "expect": "login: "},
    {"send": "admin", "expect": "Password:"},
    {"send": "<password>", "expect": "> "},
    {"send": "show version", "expect": "> ", "clear_first": true}
])
```

Capture and transcript

“Record this session” starts a capture: **raw** writes received bytes exactly as they arrive, a terminal log; **annotated** writes timestamped lines marked TX or RX, which is what you want when debugging a protocol. `capture_stop` reports the file. Independently, the last 256 KB received on every connection is kept as a transcript, so *“show me what the router printed earlier”* works even after those bytes were consumed by a read. MCP clients that browse resources see the same transcript as `serial://transcript/<name>` .

Read-only mode and idle auto-close

Two environment variables in the server's config entry: `SERIAL_CONSOLE_READ_ONLY=1` makes the server refuse anything but read-style commands (`show` , `display` , two-letter CAT reads such as `ID;` , CI-V read frames) and refuse control-line changes; it is the way to let a cautious admin try the tool on a production console.

`SERIAL_CONSOLE_IDLE_MINUTES=15` closes a port nobody has used for that long, so a forgotten connection stops blocking WSJT-X. Both are off by default.

Clean Text, Keys, and a Real Screen

A terminal program offers a menu of emulations: VT100, VT220, Xterm, ANSI, Linux, Wyse, TN3270. An agent needs three things from that list, and this chapter is those three. Everything else on the menu is either the same family under another name or a 1980s terminal nobody has plugged in for twenty years.

Three terminal modes

MODE	WHAT THE AGENT SEES	USE IT FOR
dumb	The bytes exactly as received. The default.	CAT, CI-V, rotators, GPS, Arduino, and most CLIs
ansi	Colour and escape sequences stripped as they arrive; a bare carriage return overwrites the line, backspace moves left, erase-line and cursor moves are honoured. A progress bar reads as its final state, a coloured prompt as plain text, and <code>read_until_prompt</code> matches through the colour codes.	Linux shells, coloured router prompts, anything that prints <code>ESC[32m</code> noise. The console presets set it.
xterm · vt100	All of the above plus a real character grid, 80×24 unless you say otherwise, updated continuously. The <code>screen</code> tool returns the grid as rows of text with the cursor position, and the terminal answers “who are you” queries as a VT100 so full-screen programs behave.	BIOS setup over a serial console, RAID and BMC consoles, menu-driven switches (older ProCurve, Netgear), <code>vi</code> , <code>top</code> . The <code>screen-console</code> preset.

Choose the mode on `connect(terminal=...)` or let the preset. The raw bytes are always kept in the capture file and in the screen model; only what the read tools return is cleaned. The screen model needs the `[screen]` extra (`pip install 'serial-console-mcp[screen]'`); the installers include it. Without it, `xterm` quietly runs as `ansi` and says so.

Keys by name

`send_text` types a line. `send_keys` presses keys: “hit Ctrl-C, that ping is still running”, “press Tab to complete it”, “arrow down twice and Enter”. Names are the obvious ones: `ctrl-c` through `ctrl-z`, `esc`, `tab`, `enter`, `backspace`, `delete`, `up`, `down`, `left`, `right`, `home`, `end`, `pgup`, `pgdn`, `f1 – f12`, a single character, or `text:...` for a literal run. They are encoded the way an xterm sends them, which is what every current device expects. Nothing is read; look at the reply with `read_until_prompt`, `read_available` or `screen`.

THE FULL-SCREEN LOOP

Connect with the `screen-console` preset, then alternate: press a key, read the screen, decide, press the next. Read the screen after **every** keypress; a menu that didn't move is the most common way an agent gets lost in a BIOS. The skill says the same thing to the agent.

WHAT IS DELIBERATELY NOT HERE

TN3270 is IBM mainframe over telnet, not serial. SCOANSI, Wyse 50/60 and Televideo are proprietary terminals for old SCO and Unix hosts with no maintained emulator; the rare host that still wants one usually accepts VT100. If you meet such a host, `dumb` with a capture file is the honest fallback.

Read-only mode and keys

In read-only mode, named keys are allowed (interrupting and navigating never change configuration by themselves) but single characters and `text:` runs are refused, since those are typing.

Worked Sessions

Five sessions, one per family, showing what you say, what the agent sends, and what comes back. Device replies are representative, not recordings.

1 · Juniper craft port: log in and read a version

9600 8N1 · LF · read_until_prompt

You: "Connect to the CP210x port at 9600, log in as admin, and show me the version."

TOOL list_serial_ports → /dev/cu.SLAB_USBtoUART – CP2102N USB to UART Bridge

TOOL connect(port="/dev/cu.SLAB_USBtoUART", preset="juniper-craft", name="r1") → Connected 'r1' ... 9600 (8N1, flow control none). Line ending CR, prompt '[#>%] ?\$' regex.

The agent asks you for the password rather than inventing one; you approve the expect call, which shows every step, like any other.

TOOL expect(steps=[{send:"", expect:"login: "}, {send:"admin", expect:"Password:"}, {send:"<pw>", expect:"> "}, {send:"show version", expect:"> ", clear_first:true}])

READ Step 1: sent ''+CR; matched 'login: ' after 41 bytes: r1 (ttyd0) login:

READ Step 2: sent 'admin'+CR; matched 'Password:' ... Step 3: matched '> '

READ Step 4: sent 'show version'+CR; matched '> ' after 212 bytes:\nHostname: r1\nModel: srx320\nJunos: 21.4R3-S4.15\n...\nadmin@r1>

WHY IT WORKED The preset supplied CR and the prompt regex; each step named the exact next prompt; clear_first made the last reply begin at the echo. One approval, one round trip.

2 · Kenwood-style CAT: identify the rig and read the VFO

38400 8N1 · no line ending · query_text

You: "Connect to the Kenwood at 38400 as rig and tell me what it is and where it's tuned."

TOOL connect(port="/dev/cu.usbserial-1420", preset="kenwood-cat", name="rig", baud=38400) → ... Line ending NONE, prompt ';'.

SEND query_text("ID;") ← no line ending, reads until ';' – both from the preset

READ Matched prompt ';' after 6 bytes: ID021;

SEND query_text("FA;")

READ Matched prompt ';' after 14 bytes: FA00014074000;

TOOL cat_parse("ID021;FA00014074000;") → Kenwood TS-590S · VFO A frequency: 14.074000 MHz

Agent: "It's a TS-590S on 14.074 MHz." To QSY: cat_build("FA", frequency_mhz=14.070) → FA00014070000; for query_text.

WHY IT WORKED The preset supplied both things the dialog never asks for: no line ending (Kenwood parsers reject a CR) and ';' as the prompt, so each reply returned the instant it was complete; cat_parse turned the codes into a model and a frequency.

3 · Icom CI-V: read the operating frequency in hex 19200 8N1 · send_hex → read_available

You: "Ask the IC-7300 for its frequency. CI-V address 94."

```
TOOL connect(port="/dev/cu.usbmodem14201", preset="icom-civ") → 19200 8N1, flow control none
TOOL civ_build("03", rig="94") → FE FE 94 E0 03 FD (read operating frequency)
SEND send_hex("FE FE 94 E0 03 FD", clear_buffer_first=true)
READ read_available(1.0) → Received 17 bytes Hex: fe fe 94 e0 03 fd fe fe e0 94 03 00 00 07 14 00 fd
TOOL civ_parse("fe fe 94 e0 03 fd fe fe e0 94 03 00 00 07 14 00 fd")
READ Frame 1: ... → read operating frequency (echo of our own command; the rig's reply follows)
      \nFrame 2: to E0 from 94 cmd 03 data [00 00 07 14 00] → read operating frequency: 14.070000 MHz
```

Agent: "The IC-7300 is on 14.070000 MHz." To QSY: civ_freq(mhz=14.074) gives the data bytes for civ_build("05", ...).

WHY IT WORKED Binary protocol, so the hex view is the answer and civ_parse does the BCD arithmetic; the rig's USB interface echoes, and the parser labels the echo so it isn't mistaken for a reply. XON/XOFF stayed off, so the 0x14 and 0x07 bytes arrived untouched.

4 · Arduino sensor: watch a stream, then stop it 115200 8N1 · LF · read_available

You: "Connect to the Arduino at 115200 and show me a few seconds of what it's printing."

```
TOOL connect(port="/dev/cu.usbmodem2101", preset="arduino", baud=115200)
      Opening the port toggles DTR and resets the board; the first bytes are its boot banner. To reset it again later:
      pulse_line("DTR", ms=100, level=false).
READ read_available(3.0) → Received 118 bytes Text: 'BME280 ready\n23.4 C 1012.8 hPa 41%\n23.4 C
      1012.8 hPa 41%\n23.5 C ...'
TOOL status → Reader running; 87 bytes buffered
TOOL disconnect → Disconnected from /dev/cu.usbmodem2101.
```

WHY IT WORKED A device that talks on its own has no prompt and no query; read_available with a longer timeout captures a window (capture_start would keep all of it in a file), and disconnect frees the port for the Arduino IDE.

5 · Rotator: read the heading, turn to Europe, confirm `9600 8N1 · CR · yaesu-rotator preset`

You: "Where's the beam pointing? Turn it to 45 degrees and tell me when it's there." (the rig is still open as `rig`)

TOOL `connect(port="/dev/cu.usbserial-AB12", preset="yaesu-rotator", name="rotator")` → Connected
'rotator' ... Other open connections: rig.

SEND `query_text(rotator_build("azimuth"))` → sends "C"+CR, reads until "\r"

READ Matched prompt '\r' after 6 bytes: +0180 → rotator_parse: azimuth 180° (GS-232A)

SEND `send_text(rotator_build("move", azimuth=45))` → "M045"+CR (moves reply nothing)

The agent waits, then re-reads the position until it stops changing.

READ `query_text("C")` → +0045 → azimuth 45°

Agent: "Beam is on 045. The rig connection is untouched; say the word and I'll read its frequency."

WHY IT WORKED Two ports open at once, each with its own preset; the rotator's CR line ending and "\r" reply terminator came from the preset; a move is confirmed by reading, never assumed.

When It Doesn't Answer

Almost every silent port is one of six things. The server's own messages point at most of them; this table is the rest.

SYMPTOM	MOST LIKELY CAUSE	WHAT TO SAY
No serial ports found	Charge-only USB cable, missing driver, device off. Windows: nothing under Ports (COM & LPT).	Swap the cable, then “list the ports again”
Could not open ... already in use	Another program holds the port. On a Mac or Linux the message now names it (<code>port_in_use_by</code>).	Close it, then “try connecting again”
Port name wasn't found	Names change when you replug; the remembered port is stale.	“List the ports and connect to the right one”
Garbage characters, <code>␣</code>, box drawing	Wrong baud rate. Occasionally wrong data bits or parity.	“Detect the baud rate” · “reconnect at 115200”
Echo comes back, no reply	Wrong line ending for this device, or a CLI that needs a return to wake.	“Send a return first” · “use LF instead of CR”
Prompt not seen, partial output returned	The prompt string doesn't match: missing trailing space, hostname changed, or <code>--More--</code> paging.	“The prompt is admin@r1> with a space” · “answer <code>--More--</code> with a space”
Output full of <code>ESC[0m</code>, <code>[32m</code>, box-drawing junk on a shell	Colour and cursor escape sequences shown raw: the connection is in <code>dumb</code> mode.	“Reconnect with terminal ansi” (or a console preset)
A command won't stop; the prompt never returns	ping, monitor, a pager or a hung process is still running.	“Press Ctrl-C” · “press q”
Menu screen looks like scrambled fragments	A full-screen program is repainting with cursor addressing; line tools can't show it.	“Reconnect as xterm and show me the screen”
Read-only mode refused to send	<code>SERIAL_CONSOLE_READ_ONLY</code> is set and the command isn't a read.	Unset it in the config entry, or widen <code>SERIAL_CONSOLE_ALLOW</code>
Wrong device answered	Several ports open; the command went to the current one.	“Send that to the rotator” · “what's the status?”
Write failed: Write timeout	RTS/CTS is on and the cable doesn't carry CTS.	“Reconnect without hardware flow control”
Background reader has stopped	Adapter unplugged or port taken by another program mid-session.	“Disconnect and reconnect”

Claude says it has no serial tools Claude Desktop wasn't fully quit after install, or the config entry is missing.

Quit with `⌘Q` / tray Quit, reopen, new chat

TWO QUESTIONS THAT SORT MOST OF IT

"What's the status?" tells you whether a port is open, at what settings, whether the reader is alive, and how much is buffered. *"Read whatever's waiting"* shows raw bytes as text and hex, which is how you tell a baud-rate problem (garbage) from a line-ending problem (silence).

Tool Reference

Parameters as the agent sees them. Defaults in brackets. Every tool that acts on a port takes `connection` [current].

TOOL	PARAMETERS	RETURNS
<code>list_serial_ports</code>	none	Ports with description, hardware id, open-here marker
<code>list_presets</code>	none	Each preset's settings and notes
<code>connect</code>	<code>port · preset · name · baud [9600] · bytesize [8] · parity N E O [N] · stopbits [1] · rtscts [off] · xonxoff [off] · line_ending [CR] · prompt · prompt_regex · terminal dumb ansi vt100 xterm [dumb] · cols [80] · rows [24]</code>	Settings echoed back, preset notes, other open ports
<code>reconnect_last</code>	<code>name [most recent]</code>	Same as connect, from the remembered settings
<code>disconnect</code>	<code>connection · all_connections [false]</code>	Confirmation; always releases the port
<code>send_text</code>	<code>data · line_ending [connection's] · clear_buffer_first [false]</code>	Byte count sent
<code>send_hex</code>	<code>hex_bytes · clear_buffer_first [false]</code>	Bytes sent, as hex
<code>read_until_prompt</code>	<code>prompt [connection's, else ends-with #>\$%] · timeout [10 s] · regex · auto_reply</code>	Output through the prompt; on timeout, whatever arrived
<code>read_available</code>	<code>read_timeout [1 s]</code>	Bytes as text and hex, after the line goes quiet
<code>query_text</code>	<code>data · line_ending · prompt [connection's; "" = idle] · read_timeout [5 s]</code>	The reply
<code>expect</code>	<code>steps [{send, send_hex, line_ending, expect, regex, timeout, clear_first}] · auto_reply · stop_on_timeout [true]</code>	Each step's result
<code>clear_buffer</code>	none	Bytes discarded
<code>send_keys</code>	<code>keys [ctrl-c, esc, tab, enter, up, f1, "x", text:...]</code>	Confirmation of what was pressed
<code>screen</code>	<code>reset [false]</code>	The xterm/vt100 grid as text, with cursor position
<code>status</code>	none	Every open port: settings, reader, buffer, lines, capture; or what is remembered
<code>set_lines</code>	<code>dtr · rts</code>	Resulting DTR/RTS

pulse_line	line DTR RTS · ms [100] · level [true]	Confirmation, line restored
send_break	ms [250]	Confirmation
capture_start	path [auto] · format raw annotated [raw]	The file being written
capture_stop	none	File and bytes written
get_transcript	last_bytes [4000]	Tail of everything received (rolling 256 KB); also the resource <code>serial://transcript/{name}</code>
port_in_use_by	port	Process holding it (macOS/Linux)
detect_baud	port · probe [""] · probe_line_ending [CR] · candidates · settle [0.5 s]	Rates ranked by readable reply, best guess
cat_build	command · value · frequency_mhz · flavor kenwood elecraft yaesu [kenwood]	The ';' string, with its meaning
cat_parse	reply · flavor [kenwood]	Each reply decoded: rig model, MHz, mode, IF fields, ?/E/O errors
rotator_build	action azimuth position elevation move move_azel stop ... speed · azimuth · elevation · speed	The GS-232 command
rotator_parse	reply	Azimuth/elevation in degrees, or the rejection
civ_build	command · data · rig [94] · controller [E0]	Frame as hex, with its meaning
civ_parse	hex_bytes · controller [E0]	Each frame decoded: echo or reply, frequency, mode, OK/NG, PTT
civ_freq	mhz bcd_hex	The other representation

Behaviour worth knowing

- **Non-ASCII text** in `send_text` is sent as `?` ; use `send_hex` for raw bytes.
- **Display decoding** is UTF-8 with replacement characters; prompt matching is byte-exact, so binary noise before a prompt does not shift the match.
- **Write timeout** is two seconds; **reader poll** is a tenth of a second; **idle settle** for prompt-less reads is 150 ms; the **receive buffer** holds 4 MB per port and the **transcript** 256 KB.
- **Remembered connections** live in `last_connection.json` under `~/Library/Application Support/serial-console-mcp/` (Mac), `%APPDATA%\serial-console-mcp\` (Windows), `~/.config/serial-console-mcp/` (Linux); auto-named captures go in a `captures` folder beside it.
- **Terminal modes** only change what the read tools return; the capture file and the screen model always get the raw bytes. The escape stripper is chunk-safe: a sequence split across two reads is held until complete.
- **The operating skill** in `skills/serial-console/SKILL.md` carries this guide's rules for the agent.

About & Provenance

This is an experimental 0.3.0. The guide says what the server does and what has been verified; it does not claim more.

serial-console-mcp began as a generic console tool for network craft ports and radio gear and was rewritten for this release around the background-reader model in chapter D. The 0.3.0 code is verified by a 73-case test suite driving a simulated serial port, by a live MCP transport handshake, and by continuous integration on Linux, macOS and Windows. The worked sessions in chapter E are representative walk-throughs of the tool flow, not recordings from hardware; a live-port pass is the next milestone, and this guide will say so when it has happened.

RESOURCE	WHERE
serial-console-mcp	github.com/sbrunner-atx/serial-console-mcp
Installers	GitHub Releases · SerialConsoleMCP-Setup.exe · SerialConsoleMCP-0.1.1.pkg
PyPI	pypi.org/project/serial-console-mcp · uvx serial-console-mcp
Server and tests	src/serial_console_mcp/ (server, presets, terminal, cat, civ, rotator) · tests/test_serial_console.py
Operating skill	skills/serial-console/SKILL.md
Live-port harness	_hosttest/run_live.py <steps.json>
This guide's sources	docs/brand/ (HTML + CSS, WeasyPrint)
Sibling servers	fldigi-mcp · wsjtx-mcp · n3fjp-mcp · mcp-host-bridge

Corrections and additions are welcome — open an issue on the repository.

73

This is private, personal work by Stefan Brunner, AE5VG — built for amateur radio operators and anyone else with a console cable. Not affiliated with any equipment vendor or any employer. MIT licensed. GL es 73 de AE5VG sk.