FUJINET ENGINEERING SERIES

Connecting an Emulator to FujiNet-PC

How to give a vintage-computer emulator a real FujiNet — by carrying the machine's peripheral bus over a socket to the FujiNet firmware running on your desktop.

Worked example: the Coleco ADAM emulator ADAMEm ↔ fujinet-pc

Revision 1 · June 2026
The FujiNet Project

About this guide

This manual is for the author of a vintage-computer **emulator** who wants to add FujiNet support: to let software running inside the emulator mount disks, reach the network, and use the FujiNet device suite exactly as it would on real hardware. The technique is to run the **real FujiNet firmware** as a desktop program (`fujinet-pc`) and connect the emulator to it by carrying the emulated machine's peripheral bus over a TCP socket — the **Bus over IP** model pioneered by Atari's NetSIO.

It is written as a worked example. We take a specific, real emulator — **ADAMEm**, a Coleco ADAM emulator — and a specific FujiNet build — **fujinet-pc** compiled for the ADAM target — and walk through every change that made them talk, explaining the protocol, the seams, and the hard-won pitfalls. The assumption is that you know your own emulator's code; the goal is to teach the reasoning so you can repeat it on the machine you maintain.

Every packet field, register value, timeout, and source excerpt was transcribed from the live project sources listed below, not from secondary documentation. Where a path is still experimental, the text says so plainly.

Canonical sources

adamem_sdl	The Coleco ADAM emulator being adapted — `AdamNet.c/.h` , `Coleco.c` , `ADAMEm.c`
fujinet-pc-adam	FujiNet firmware built as a PC app (ADAM target) — `NetAdamNet` , `lib/bus/adamnet`
fujinet-firmware	The canonical AdamNet bus + device firmware the PC build derives from
fujinet-emulator-bridge	The Atari NetSIO precedent for "Bus over IP"

Contents

PART I

The Idea

What it means to give an emulator a real FujiNet, why you would want to, and the Bus-over-IP model that makes it possible. Read this part before touching your emulator's code.

CHAPTER 1

Introduction

FujiNet is a network and storage peripheral for retro computers. To a 1980s machine it looks like a fast disk drive, a printer, an RS-232 modem, a real-time clock, and a handful of network adapters; behind that façade an ESP32 does the real work — fetching files over Wi-Fi, mounting disk images from a TNFS server on the other side of the world, parsing JSON, serving a configuration program. There is a FujiNet for the Atari 8-bit, the Apple II, the Coleco ADAM, the Tandy Color Computer, the IBM PC, and more.

This manual is about a different kind of FujiNet user: not a person with a real machine on their desk, but an **emulator**. If you maintain an emulator for one of these platforms, you can give the software running inside it a genuine FujiNet — not a partial reimplementation of a few disk commands, but the **actual FujiNet firmware**, running as an ordinary program on the same PC, reachable over a socket. The emulated machine mounts real disk images, browses real TNFS hosts, and runs the same `CONFIG` program a real FujiNet boots. Everything the hardware does, the emulator now does too, because it is talking to the same code.

1.1 What you are about to build

The trick has three pieces:

Figure 1. The whole idea on one line. Your emulator already pretends to be the machine; it now forwards the machine's peripheral-bus traffic over a local socket to the unmodified FujiNet firmware compiled as a desktop application.

The emulator keeps emulating the CPU, video, sound, and keyboard. The one thing it stops pretending to be is the **peripheral bus** — the wire that, on real hardware, connects the computer to its disk drives and to the FujiNet cartridge. Instead of synthesising replies to disk commands itself, the emulator carries those bus transactions, byte for byte, over a TCP socket to `fujinet-pc`, which answers them exactly as the hardware FujiNet would.

We will build this for a specific pair of programs and explain every line of the seam:

`ADAMEm` (`adamem_sdl`)	A mature Coleco ADAM emulator. It already emulates the ADAM's CPU, VDP, sound, keyboard, tape, and disks. We add a bridge that turns it into the **AdamNet master** for a chosen set of device IDs.
`fujinet-pc` (ADAM target)	The FujiNet firmware, compiled to run on Linux/macOS/Windows instead of an ESP32. It already speaks AdamNet; we changed only the **transport** so the bus arrives over a socket instead of a UART.

Table 1. The two halves of the worked example. Both are real, both are in the workspace, and both have been adapted already — this manual reconstructs how.

1.2 Who this manual is for

You are the author or a maintainer of an emulator for some vintage machine, and you know your own codebase. You do not need to know FujiNet's internals, AdamNet, or the ADAM; those are taught here as we go. What you **do** need is a clear picture of where, inside your emulator, the emulated software talks to its peripherals — because that is the seam we will cut. Chapter 6 is devoted to finding it.

The worked example is the Coleco ADAM, but the manual is structured so the ADAM-specific parts are clearly separated from the parts that transfer to any machine. Each chapter in Part III ends with a short "For your emulator" note that restates the lesson without the ADAM.

1.3 How to read it

The book is in five parts:

Part I — The Idea the model and the motivation (you are here).

Part II — The Bus You Will Speak AdamNet and its wire protocol, taught from scratch. This is the protocol the emulator's master must reproduce.

Part III — Adapting the Emulator the worked example, change by change — finding the seam, the socket, the master state machine, device routing, the boot handshake, and the build.

Part IV — Getting It Right the pitfalls. Five of them were real bugs fixed after the first version worked; the sixth is the peripheral's point of view. Skipping this part is the difference between "it boots" and "it boots reliably over a laggy TNFS link without the music stuttering."

Part V — Your Turn a platform-agnostic checklist, how to run `fujinet-pc` as your reference peripheral, and reference appendices.

> **NOTE**
> Throughout, file names in `monospace` refer to real files. On the emulator side they live in `adamem_sdl/`; on the FujiNet side, in `fujinet-pc-adam/`. Commit hashes such as `d0f79b0` refer to the actual history of `adamem_sdl`, so you can read the change as a diff if you prefer patches to prose. Appendix E maps every change to its file.

CHAPTER 2

Why give your emulator a FujiNet

Adding a network peripheral to an emulator is work. Before the how, the why — because the reasons shape several design decisions later, and because "use the real firmware over a socket" is not the only option you could have chosen.

2.1 You ship the real firmware, not a clone

The naïve approach to "network support in an emulator" is to reimplement the peripheral inside the emulator: catch the disk commands, open a local file, maybe bolt on an HTTP fetch. People have done this for decades. It always drifts. The real device gains a command, fixes a quirk, changes a status byte; the emulator's hand-written clone does not, and software that works on hardware mysteriously fails under emulation (or worse, the reverse).

Bus over IP avoids the clone entirely. The emulator runs **the same firmware the hardware runs**, built for a different CPU. When FujiNet adds a feature, the emulator gets it for free the next time it links against a newer `fujinet-pc`. There is exactly one implementation of "what FujiNet does," and both the hardware and the emulator use it.

> **TIP**
> This is the single strongest argument for the socket approach over a built-in clone, and it is worth keeping in mind when a pitfall later tempts you to "just special-case this one command in the emulator." Every such special case is the first crack of the drift you were trying to avoid.

2.2 What your users get

For the person using your emulator, a real FujiNet means:

- **Disks from anywhere.** Mount `.dsk` / `.ddp` images from a local folder, a microSD image, or a TNFS server on the public internet — the same hosts real FujiNet users browse. A piece of homebrew posted to a TNFS share boots in the emulator with no extra packaging.
- **The genuine `CONFIG` experience.** The FujiNet configuration program — host lists, drive slots, the Fuji menu — runs inside the emulator exactly as on hardware, because it is the same program reading the same device.
- **Live network apps.** Clients that fetch weather, read a Mastodon timeline, play network games, or open a TCP/Telnet session work under emulation. The worked example boots `fujinet-connect-four.ddp` and an ISS tracker that pulls the station's position over HTTP — in the emulator.
- **A safe place to learn.** New users can explore FujiNet without owning the cartridge or the vintage machine, then move to hardware with everything already familiar.

2.3 What the FujiNet project gets

There is a second audience: the FujiNet developers themselves. An emulator wired to `fujinet-pc` is the most productive FujiNet development environment there is.

- **Edit-compile-run in seconds.** `fujinet-pc` is an ordinary desktop program. Change device firmware, rebuild, relaunch, and the emulated machine exercises it immediately — no flashing an ESP32, no reseating a cartridge.
- **A real debugger.** `gdb`, `valgrind`, and AddressSanitizer all work on `fujinet-pc`. Several bugs were found this way the first time the ADAM code ran on the PC at all: a NULL-`FILE` crash in the `.ddp` media handler, a char-narrowing init bug in the ROM media type, and a bad network response buffer. (See `fujinet-pc-adam` commit `b0e57228b`.)

- **CI without hardware.** The emulator-plus-firmware pair can boot a disk and assert on the result in a headless CI job — regression coverage that hardware-in-the-loop testing cannot match for cost or speed.

2.4 And you, the emulator author

Finally, the benefit that is easy to overlook: you do not have to become a FujiNet expert, and you do not have to maintain a network stack. The emulator gains Wi-Fi, TNFS, HTTP, JSON parsing, SSH, Telnet, a clock, and a printer — none of which you wrote, and none of which you will be on the hook to keep working. Your responsibility ends at the bus. That is a remarkably small surface for a remarkably large feature.

> **IMPORTANT**
> The whole approach rests on one fact: **the FujiNet firmware can be compiled for the PC.** For ADAM, Atari, Apple II, CoCo, and the RS-232 platforms, that work is already done — `fujinet-pc` builds today. If your platform's device code is not yet PC-buildable, that is the prerequisite, and it is a FujiNet-firmware task, not an emulator task. Chapter 19 points at what is involved.

CHAPTER 3

The Bus-over-IP architecture

Everything in this manual is an elaboration of one model. This chapter states it plainly; the rest of the book is detail.

3.1 The roles: master and peripheral

On a real vintage machine, the CPU does not talk to a disk drive directly. It talks to a **bus** — a shared electrical connection with a protocol — and one side of that bus is in charge. On the Atari it is the SIO bus; on the CoCo, the serial/DriveWire link; on the ADAM, **AdamNet**. The computer (or a dedicated controller chip inside it) is the **master**: it issues commands and expects timely replies. The disk drives, printer, and FujiNet are **peripherals**: they listen, and they answer when addressed.

Bus over IP preserves these roles exactly, and assigns them across the two programs:

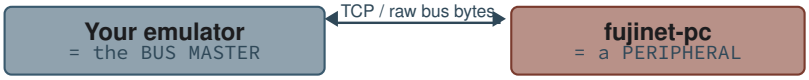

Figure 2. The emulator becomes the bus master — the part of the real machine that drives the peripheral bus. `fujinet-pc` is a peripheral on that bus, answering when its device IDs are addressed, just as the cartridge does on hardware.

This is the central design decision, and it is worth dwelling on. The emulator does not become a **client** of FujiNet in any application sense. It becomes the **bus controller**. It sends the same command bytes the real machine's controller sends, in the same order, with the same timing expectations, and it interprets the replies with the same state machine. The firmware on the other end cannot tell — and does not care — whether the commands came from a 6801 over a one-wire UART or from an emulator over a socket.

> **NOTE**
> This is why the manual spends two chapters (4 and 5) teaching you the ADAM's wire protocol before any socket code appears. To be the master, the emulator must **speak the bus**. The socket is the easy part; the protocol is the work.

3.2 The transport: a socket carrying raw bus bytes

What travels over the TCP connection is not a high-level RPC, not JSON, not a custom "emulator protocol." It is the **raw bytes of the peripheral bus** — exactly the octets that would be on the wire between the machine and the cartridge. The socket is a dumb pipe standing in for a piece of wire.

This matters because it keeps the firmware side almost unchanged. On real hardware, FujiNet reads AdamNet bytes from a UART; on the PC, it reads the same bytes from a socket. Only the few lines that fetch and emit bytes had to change — the entire command-processing, device, and media stack above them is untouched. We will see exactly that boundary in Chapter 17.

3.3 Who connects to whom

A TCP connection has an asymmetry the bus does not: someone must **listen** and someone must **connect**. We resolve it like this:

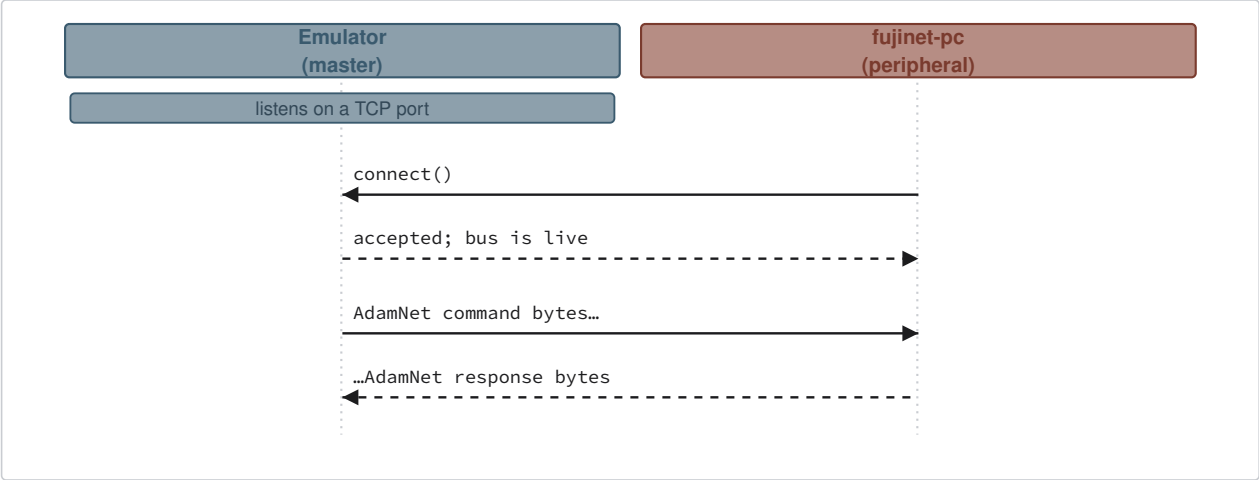

Figure 3. The emulator is the TCP **server** (it `listen()`s); `fujinet-pc` is the TCP **client** (it `connect()`s in and retries quietly until the emulator is up). Once connected, raw bus bytes flow both ways for the life of the session.

The choice is deliberate and has a practical payoff: `fujinet-pc` can be left running as a background service that simply keeps trying to connect, and the emulator can come and go. The firmware resolves the host once, then retries silently so a long-lived service does not spam its log (`NetAdamNet` `ensure_connected()`). We will return to the boot-timing consequences in Chapter 10 — the emulator generally wants to **wait** for the peripheral before it lets the machine boot, so the first disk scan already sees the drive.

3.4 The lineage: NetSIO

This model is not new with the ADAM. It is a direct descendant of the Atari world's **NetSIO**, used by the `fujinet-emulator-bridge` project to connect the Altirra Atari emulator to `fujinet-pc` over a UDP hub. The ADAM work mirrors it on purpose; where the two differ (TCP vs UDP, who listens), the differences are noted so an Atari-side reader is not surprised. If your platform is Atari, much of this is already built — see Chapter 19.

Bus	SIO	AdamNet
Master	Altirra (emulator)	ADAMEm (emulator)
Peripheral	fujinet-pc	fujinet-pc
Transport	UDP via a hub	direct TCP
Listener	the hub	the emulator
Default port	9997	65216

Table 2. Bus over IP across two platforms. The roles are identical; the plumbing differs in ways that do not affect the model.

With the model in hand, we now turn to the bus itself.

PART II

The Bus You Will Speak

To stand in for the machine's bus controller, your emulator must speak the machine's peripheral bus fluently. This part teaches AdamNet — its shape, its byte format, and the exact handshakes the master runs — using the FujiNet firmware's own definitions as the reference. Even if your platform is not the ADAM, read it for the pattern: every platform's bus has an equivalent of each idea here.

CHAPTER 4

AdamNet: the ADAM's peripheral bus

The Coleco ADAM is unusual among 8-bit machines: its peripherals are not memory-mapped cards or a simple disk controller, but a small **network**. Understanding its shape is the foundation for everything the emulator's master does.

4.1 A network of small computers

Inside the ADAM, the Z80 that runs user software is not in charge of the peripherals at all. A second processor — a 6801 **network master** — owns a serial bus called **AdamNet**, and every peripheral (the disk drives, the tape drives, the keyboard, the printer) is itself a little 6801-based node hanging off that bus. The Z80 asks the master to do things; the master runs the bus transactions.

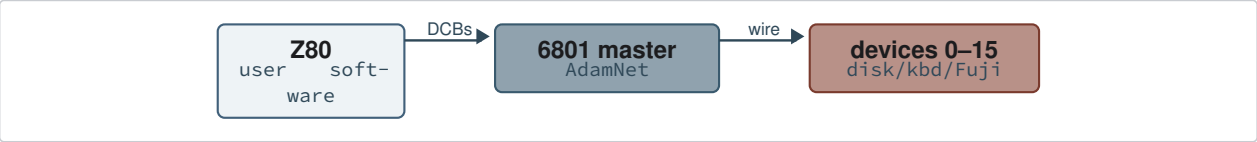

Figure 4. The ADAM's two-tier model. User code on the Z80 fills in **Device Control Blocks** in shared RAM; the 6801 network master turns those into AdamNet wire transactions with the addressed device.

There are two consequences for us, and they shape the whole adaptation:

1. **The seam in the emulator is the master.** ADAMEm does not have to emulate each peripheral node; it has to emulate the **6801 master** well enough to run the wire protocol. When we forward a device to FujiNet, ADAMEm becomes the master for that device and runs real AdamNet transactions over the socket.
2. **There are two protocols, not one.** Between the Z80 and the master sits the **DCB** protocol (Chapter 9). Between the master and the devices sits the **AdamNet wire** protocol (Chapter 5). FujiNet lives on the wire side, so the emulator must translate from one to the other. That translation is the core of the bridge.

4.2 Devices are numbered 0–15

Every AdamNet device has a 4-bit ID. The IDs that matter for FujiNet:

`0x01`	Keyboard	No — stays local to the emulator
`0x02`	ADAMNet printer	Yes (char device)
`0x04` – `0x07`	Disk drives 1–4	Yes (block devices)
`0x08`	Tape	No — stays local
`0x09` – `0x0E`	Network adapters	Yes (char devices)
`0x0F`	The Fuji "gateway" device	Yes (char device)

Table 3. AdamNet device IDs and the FujiNet routing used in the worked example. The forwarding set is a bitmask in `AdamNet.c`; keyboard and tape are deliberately kept local so the emulator's own hardware keeps working.

The high nibble of every bus byte is a **command** or **response** code; the low nibble is the device ID it is addressed to. That single-byte framing is the first thing to learn about the wire, and it is the subject of the next chapter.

> **NOTE**

> **For your emulator.** Your platform almost certainly does not have a 6801 network. But it has the same two layers in some form: a high-level way the OS asks for disk/printer/serial I/O (the ADAM's DCBs; the Atari's CIO/SIO call blocks; an MS-DOS `INT`), and a low-level bus protocol underneath (the ADAM's AdamNet; the Atari's SIO frames). Find both. The seam you cut is between them.

CHAPTER 5

The AdamNet wire protocol

This is the protocol your emulator's master will speak over the socket. It is defined, byte for byte, in the FujiNet firmware (`lib/bus/adamnet/adamnet.h`) and mirrored in the emulator's `AdamNet.c` . Everything here is transcribed from those two files.

5.1 The byte format

AdamNet is a **half-duplex one-wire serial bus** running at 62500 baud. On the wire, control is carried in single bytes: the high nibble is the operation, the low nibble is the device.

high nibble = command/response code low nibble = device ID (0–15)

Two macros in the emulator build these bytes, and they are worth memorising because they appear in every transaction:

```
#define CMD(c,dev)  (unsigned char)(((c) << 4) | ((dev) & 0x0F))
#define RESP(r,dev) (unsigned char)(((r) << 4) | ((dev) & 0x0F))
```

So a "STATUS request to device 0x0F" is `CMD(MN_STATUS, 0x0F)` = `0x1F` , and the device's status reply is `RESP(NR_STATUS, 0x0F)` = `0x8F` .

5.2 Command and response codes

The master sends **command** codes; the device answers with **response** codes. Note the deliberate split: commands occupy the low half of the nibble space, responses the high half, so you can tell direction at a glance.

`0x0`	`MN_RESET`	Reset the device
`0x1`	`MN_STATUS`	Request a status packet
`0x2`	`MN_ACK`	Acknowledge
`0x3`	`MN_CLR`	Clear to send (CTS) — "stream me the data now"
`0x4`	`MN_RECEIVE`	Device, prepare/stage data for me
`0x5`	`MN_CANCEL`	Cancel
`0x6`	`MN_SEND`	Master is sending you a payload
`0x7`	`MN_NACK`	Negative acknowledge
`0xD`	`MN_READY`	Ready?

Table 4. Master command codes (high nibble). From `adamnet.h` , identical in the emulator's `AdamNet.c` .

`0x8`	`NR_STATUS`	Here is my 6-byte status packet
`0x9`	`NR_ACK`	Acknowledged
`0xA`	`NR_CANCEL`	Cancelled
`0xB`	`NR_SEND`	Here comes a data packet
`0xC`	`NR_NACK`	Negative acknowledge / nothing available

Table 5. Device response codes (high nibble). The firmware spells these `NM_*` ; the emulator spells them `NR_*` . Same values.

5.3 Packets: length and checksum

Beyond single control bytes, AdamNet carries **packets**. A packet sent by the master with `MN_SEND`, or by the device with `NR_SEND`, has this shape:

cmd|dev	len hi	len lo	payload (len bytes)…	checksum

- The **length** is 16-bit. There is a wrinkle worth pinning down now because it bites later: the length is **big-endian** in data packets and in the emulator's char/disk sends, but a few fixed-size sends spell it out by hand. When in doubt, read the exact transaction in Chapter 8.
- The **checksum** is a simple 8-bit XOR of every payload byte:

```
static unsigned char an_checksum (const unsigned char *buf,int len)
{
 unsigned char ck=0; int i;
 for (i=0;i<len;++i) ck^=buf[i];
 return ck;
}
```

PITFALL
It is an **XOR**, not an additive checksum. (Compare FujiNet's other buses: the FujiBus/SLIP transport on the tandem platforms uses an add-with-carry fold. Do not copy a checksum routine from another platform's code.) The firmware's `adamnet_checksum()` is the same XOR.

5.4 The status packet

A `MN_STATUS` command always elicits a fixed **6-byte** reply, and it is the simplest complete transaction on the bus — which is exactly why we will use it as a liveness probe in Chapter 10.

`0x8` |dev	len lo	len hi	devtype	status	checksum

```
int AdamNet_DiskStatus (int dev,unsigned char *status_byte)
{
 unsigned char pkt[6];
 if (an_send_byte (CMD(MN_STATUS,dev))) return -1;
 if (an_recv (pkt,6,TMO_ACK)) return -1;          /* read the 6-byte reply   */
 if (pkt[0]!=RESP(NR_STATUS,dev)) return -1;      /* must be 0x8|dev          */
 if (status_byte) *status_byte=pkt[4];            /* byte 4 = status          */
 return 0;
}
```

`devtype` distinguishes a block device (`0x01`) from a character device (`0x00`); `status` carries device-specific flags. For our purposes the transaction's real value is binary: **a well-formed 6-byte reply starting with `0x8|dev` means the device is alive and speaking AdamNet.** That is enough to tell a real FujiNet from a wrong-platform peer that happened to grab the port.

5.5 The block-read handshake

Disk reads are the most involved transaction and the one whose timing causes the most trouble, so we walk it in full. A block is **1024 bytes**. Reading one is a three-step conversation:

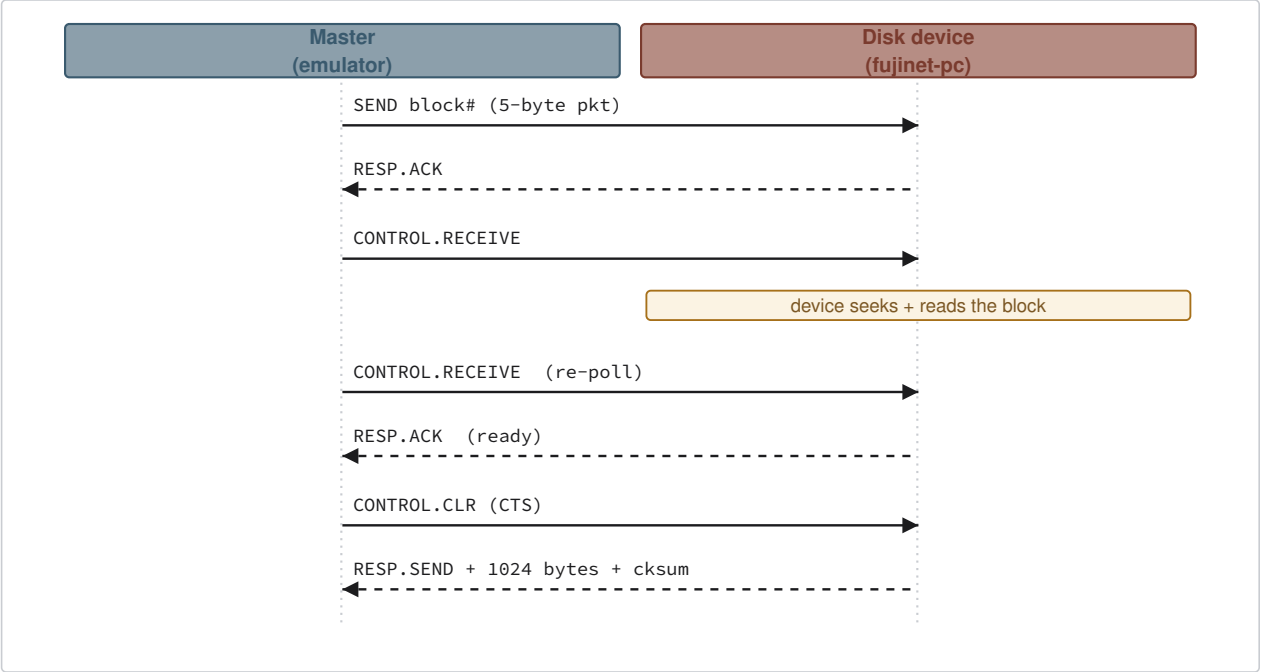

Figure 5. The disk block-read handshake. Step 1 names the block; step 2 (`RECEIVE`) tells the device to fetch it and is **re-polled** while the device is still seeking; step 3 (`CLR` /CTS) pulls the data packet. The re-poll loop in step 2 is the source of nearly every pitfall in Part IV.

In words:

1. **Name the block.** The master sends a 5-byte `MN_SEND` packet whose payload is the 32-bit block number (little-endian) plus a trailing zero. The device ACKs.
2. **Ask for it (`MN_RECEIVE`).** The device begins fetching the block. On real hardware this is a head seek; on FujiNet it may be a microSD read or a multi-second TNFS fetch across the internet. **While it is busy, it stays silent,** and the master must re-issue `MN_RECEIVE` until it finally gets a `RESP.ACK` . This re-poll loop is fundamental — it is how the bus tolerates a slow device — and getting it wrong is the subject of three different chapters in Part IV.
3. **Pull the data (`MN_CLR`).** The master sends Clear-To-Send; the device streams an `NR_SEND` data packet: the header byte, a **big-endian** length of 1024, the 1024 data bytes, and a checksum byte.

5.6 The block-write handshake

Writes are simpler — there is no seek-stall re-poll, because the master is the one supplying data:

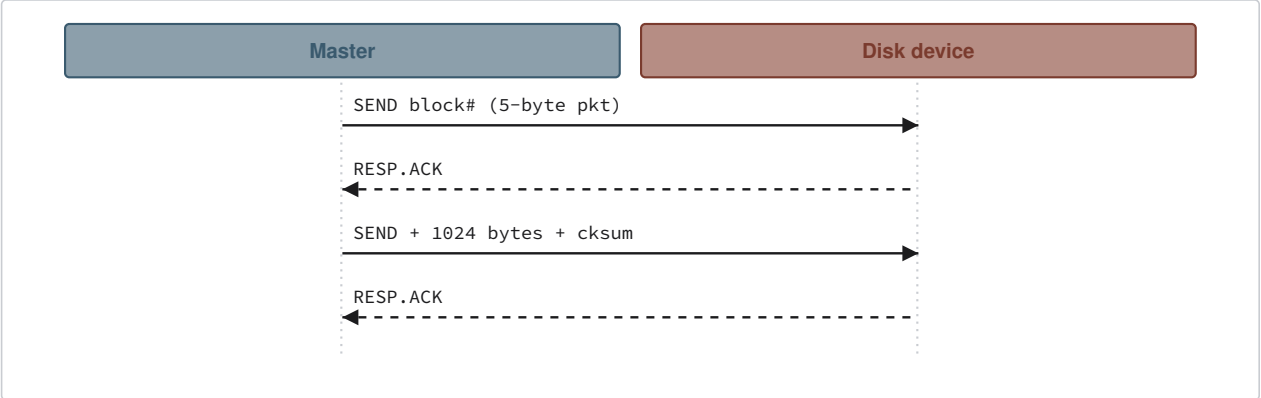

Figure 6. The block-write handshake. Name the block, then send the 1024-byte payload; the device ACKs each step.

5.7 The character-device handshake

The Fuji gateway, the network adapters, and the printer are **character** devices: variable-length request/response rather than fixed 1024-byte blocks. The pattern reuses the same codes:

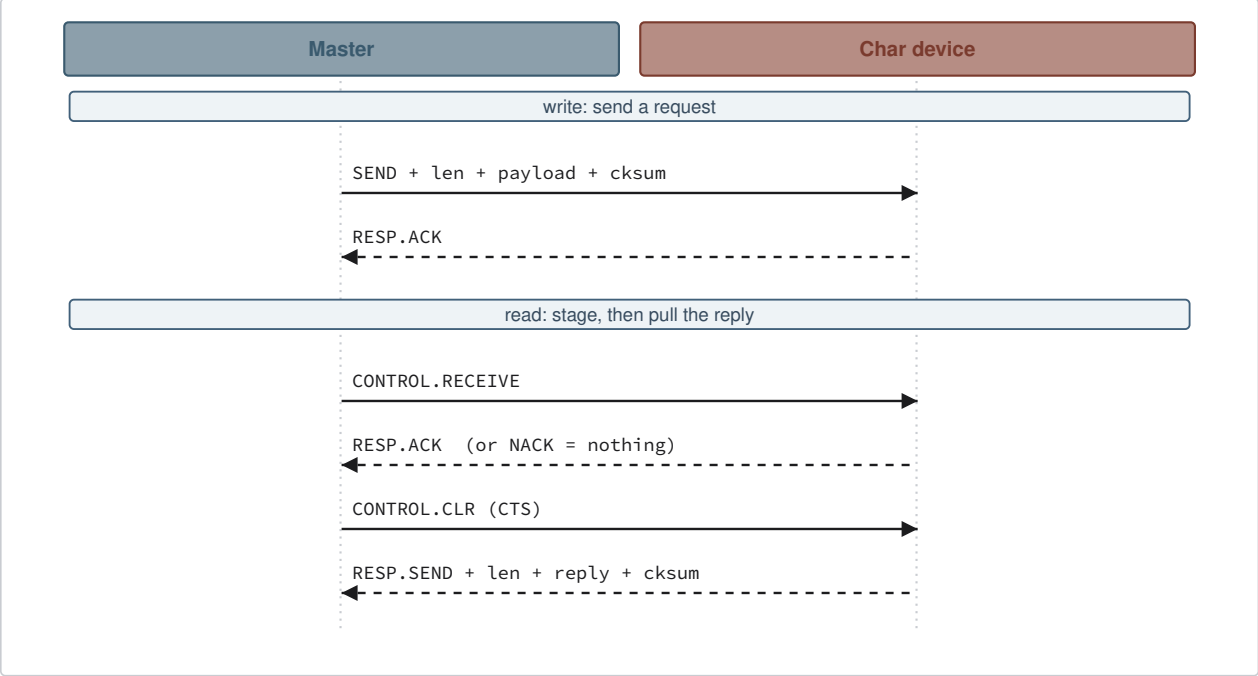

Figure 7. The character-device handshake. A read **must** issue `RECEIVE` before `CLR` : that is what tells the device to stage its pending data (e.g. a JSON query result) into its response buffer. Skip it and the device NACKs the `CLR` because its response length is still zero.

> **IMPORTANT**
> The character path is marked **experimental** in `AdamNet.h` : the DCB-op-to-wire mapping for char devices has not been fully validated against EOS behaviour. Disk and status are the proven path and are what boots disks and runs `CONFIG` . We document the char path because it is the gateway to the full FujiNet device suite, but treat it as the frontier, not bedrock.

That is the entire wire vocabulary. With it, we can read every line of the emulator's master and understand exactly what each byte on the socket means. Part III builds that master.

PART III

Adapting the Emulator

The worked example, change by change. We find the seam in ADAMEm, open a socket, build the AdamNet master state machine, route device I/O across it, handle the boot handshake, and wire it into the build. Each chapter ends with a “For your emulator” note that lifts the lesson off the ADAM.

CHAPTER 6

Finding the seam

This is the most important chapter in the book, because it is the one decision you cannot copy from the ADAM: **where, in your emulator, do you cut?** Everything else follows mechanically once the seam is right. Get it wrong and you will fight your own code forever.

6.1 What you are looking for

You want the narrowest place where the emulated software's **peripheral intent** is fully formed but has not yet been turned into fake hardware behaviour. At that point you have a clean request — "read block N of drive 2 into address A" — that you can forward whole, and a clean place to write the answer back. Cut too high (inside the OS call) and you must re-implement the OS; cut too low (inside the UART emulation) and you are reconstructing intent from bit-banged edges.

In ADAMEm, that place is glaringly obvious once you know the ADAM: it is the **Device Control Block** handler.

6.2 Device Control Blocks

Recall from Chapter 4 that the Z80 does not drive AdamNet; it asks the 6801 master to. The mechanism is a **Device Control Block** — a small structure in shared RAM. The Z80 fills in a command, a buffer address, a length, and a block number, then pokes the master; the master runs the AdamNet transaction and writes a status byte back into the DCB. ADAMEm already emulates this: a function `UpdateDCB()` in `Coleco.c` is called whenever the Z80 touches a DCB, and it dispatches on the device ID.

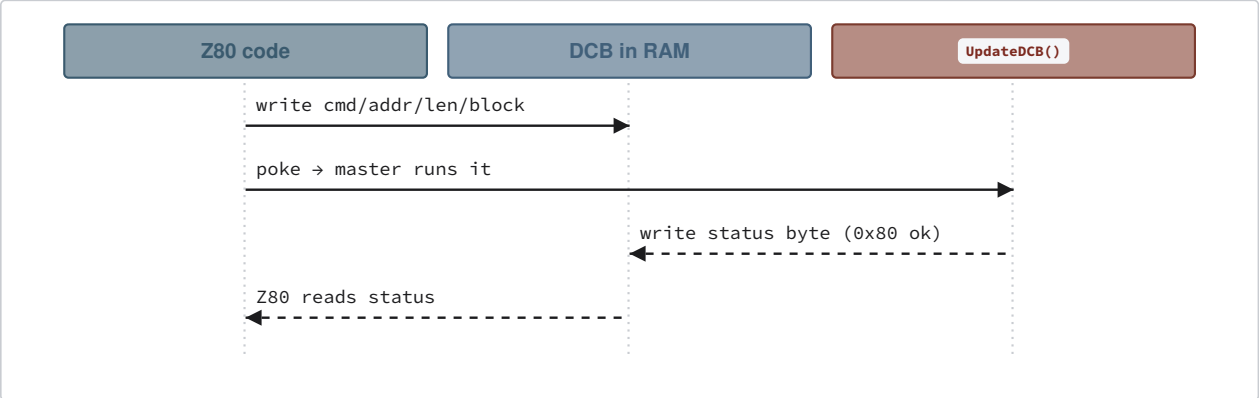

Figure 8. How ADAMEm already handles a DCB before any FujiNet code exists. `UpdateDCB()` is the emulator's stand-in for the 6801 master. This is the function we will intercept — it is the seam.

The device ID is encoded across two DCB fields, and decoding it is the first line of `UpdateDCB()`:

```
dev_id = (RAM[(DCB+9)&0xFFFF] << 4) + (RAM[(DCB+16)&0xFFFF] & 0x0F);
```

So `dev_id` is already the AdamNet device number we learned in Chapter 4. Everything we need is in scope at this one point: the device, the operation, the buffer, the length, the block. **This is the seam.**

6.3 Cutting it

The intercept is four lines, added at the top of `UpdateDCB()` before its existing `switch`:

```
dev_id=(RAM[(DCB+9)&0xFFFF]<<4)+(RAM[(DCB+16)&0xFFFF]&0x0F);
if (AdamNet_IsForwarded (dev_id) && AdamNet_Connected ())
{
 UpdateFujiNet (mode,dev_id,DCB);
 return;
}
switch (dev_id) { /* ...the emulator's own keyboard/tape/disk handling... */ }
```

That is the entire integration point on the I/O side. If the device is in the forwarding set **and** `fujinet-pc` is connected, hand the DCB to a new function `UpdateFujiNet()` (Chapter 9) and return; otherwise fall through to ADAMEm's existing emulation untouched. The two predicates matter:

- `AdamNet_IsForwarded(dev_id)` keeps keyboard (`0x01`) and tape (`0x08`) local, so the emulator's own input and cassette keep working while disks and the network go to FujiNet.
- `AdamNet_Connected()` means that **until** `fujinet-pc` **connects**, the emulator behaves exactly as it always did. The bridge is purely additive: a user who never passes `-fujinet` sees no change at all.

TIP
The "purely additive" property is worth engineering for deliberately. It means your existing users and tests are unaffected, your bug surface is bounded to the new path, and you can ship the feature disabled by default. Every predicate that guards the new code with "only if the bridge is active" is buying you that safety.

NOTE
For your emulator. Find your `UpdateDCB` . On the Atari it is the SIO device-vector dispatch; on a CoCo it is the DriveWire request handler; on MS-DOS-class emulation it might be an `INT 13h` / `INT 21h` shim. The signature of the right seam is always the same: a single function that already receives a fully-formed peripheral request and already knows where the answer goes. Add the same two-predicate guard there.

CHAPTER 7

The transport layer

With the seam identified, the first new code is the dumb pipe: a TCP listener that waits for `fujinet-pc` to connect, and helpers to push and pull bytes with timeouts. This is `AdamNet.c`'s socket layer, and it is the least platform-specific code in the whole project — if you have ever written a blocking-with-timeout socket helper, you can skim it.

7.1 The emulator listens

Per Chapter 3, the emulator is the TCP **server**. `AdamNet_Init()` opens a non-blocking listening socket:

```
int AdamNet_Init (int port)
{
 an_listen_fd=socket (AF_INET,SOCK_STREAM,0);
 setsockopt (an_listen_fd,SOL_SOCKET,SO_REUSEADDR,&on,sizeof(on));
 /* bind INADDR_ANY:port, listen(), then: */
 fcntl (an_listen_fd,F_SETFL,O_NONBLOCK);
 an_enabled=1;
 printf ("AdamNet: listening for fujinet-pc on TCP port %d\n",port);
 return 0;
}
```

`SO_REUSEADDR` lets you restart the emulator without waiting out `TIME_WAIT`; the non-blocking listener lets `AdamNet_Connected()` poll for a peer without ever stalling the emulator's main loop:

```
int AdamNet_Connected (void)
{
 if (an_conn_fd>=0) return 1;            /* already connected            */
 fd=accept (an_listen_fd,NULL,NULL);
 if (fd<0) return 0;                     /* nobody waiting (non-blocking) */
 setsockopt (fd,IPPROTO_TCP,TCP_NODELAY,&on,sizeof(on));
 an_conn_fd=fd;
 return 1;
}
```

IMPORTANT

`TCP_NODELAY` is not optional. AdamNet is a tight request/response handshake — the master sends one byte and waits for one byte. With Nagle's algorithm on, the kernel would sit on your single-byte `MN_RECEIVE` waiting for more data that never comes, adding up to 40 ms per round trip and turning a brisk handshake into a crawl. Both ends set `TCP_NODELAY`; `fujinet-pc` sets it too (`NetAdamNet::ensure_connected`).

7.2 Bytes in, bytes out

The byte-level transport is three small functions. Sending loops over partial writes and treats any hard error as a disconnect:

```
static int an_send (const unsigned char *buf,int len)
{
 int off=0,n;
```

```
 while (off<len)
 {
  n=send (an_conn_fd,buf+off,len-off,0);
  if (n>0) { off+=n; continue; }
  if (n<0 && (errno==EINTR)) continue;
  an_disconnect ();
  return -1;
 }
 return 0;
}
```

Receiving takes a **timeout** — this is the crucial parameter that the whole master state machine is built on. `an_recv()` reads exactly `len` bytes or fails after `timeout_ms`, using `select()` so it never blocks forever:

```
static int an_recv (unsigned char *buf,int len,int timeout_ms)
{
 int got=0,n;
 while (got<len)
 {
  /* select() on an_conn_fd with tv = timeout_ms */
  n=select (an_conn_fd+1,&rfds,NULL,NULL,&tv);
  if (n==0) return -1;                    /* timed out                    */
  n=recv (an_conn_fd,buf+got,len-got,0);
  if (n>0) { got+=n; continue; }
  if (n<0 && errno==EINTR) continue;
  an_disconnect ();                       /* peer closed or error         */
  return -1;
 }
 return 0;
}
```

A timeout of **zero** is special and we will lean on it hard in Part IV: it makes `an_recv()` a **non-blocking peek** — return a byte if one is already waiting, else fail immediately. That single property is what lets a disk read poll the socket without ever stalling the emulated CPU.

7.3 Draining the pipe

One more helper exists only because of a pitfall (Chapter 13), but it belongs with the transport: `an_drain()` discards any bytes already sitting in the socket without blocking.

```
static void an_drain (void)
{
 for (;;)
 {
  /* select() with a zero timeout: poll, never block */
  if (select (an_conn_fd+1,&rfds,NULL,NULL,&tv)<=0) break;
  if (recv (an_conn_fd,scratch,sizeof scratch,0)<=0) break;
 }
}
```

It only ever sees bytes the device already sent, so it is safe to call before a transaction to clear stale state. Why a healthy transaction can leave "stale" bytes behind is a story about slow disks; hold the question until Chapter 13.

> **NOTE**
> **For your emulator.** This layer is portable as-is. The only platform decisions are the framing (a stream socket carrying raw bus bytes — keep it dumb) and whether you listen or connect (match the precedent for your

> platform: the ADAM emulator listens; the Atari NetSIO hub listens). The timeout-bearing `recv` and the zero-timeout peek are the two primitives the rest of the manual assumes you have.

CHAPTER 8

The master state machine

Now the substance: the functions that actually run AdamNet transactions over the socket. These **are** the 6801 master, reimplemented in C against a TCP stream. Each maps one-to-one onto a handshake from Chapter 5, so if those diagrams are fresh, this chapter is mostly confirmation.

8.1 Naming a block

Every block transaction starts by telling the device which block. The 5-byte `MN_SEND` packet carries the 32-bit block number little-endian, plus a trailing zero byte, then a checksum; the device must ACK:

```
static int an_send_block_num (int dev,unsigned long block)
{
 unsigned char pkt[5];
 pkt[0]= block        & 0xFF;            /* 32-bit block #, little-endian */
 pkt[1]=(block >>  8) & 0xFF;
 pkt[2]=(block >> 16) & 0xFF;
 pkt[3]=(block >> 24) & 0xFF;
 pkt[4]=0x00;
 if (an_send_byte (CMD(MN_SEND,dev))) return -1;
 if (an_send_byte (0x00)) return -1;     /* length high (0x0005)          */
 if (an_send_byte (0x05)) return -1;     /* length low                    */
 if (an_send (pkt,5)) return -1;
 if (an_send_byte (an_checksum (pkt,5))) return -1;
 if (an_recv_byte (TMO_ACK)!=RESP(NR_ACK,dev)) return -1;
 return 0;
}
```

Note the hand-spelled length bytes `0x00, 0x05`. The block-number length is fixed at 5, so the code writes it literally rather than going through a length helper. This is the kind of small inconsistency you will find in any real bus implementation; transcribe it exactly rather than "tidying" it.

8.2 Reading a block (the blocking version)

The synchronous read is the clearest statement of the Chapter 5 handshake. We will **replace** it in Chapter 14 with a non-blocking version — but read this one first, because the non-blocking one is this logic turned inside out, and it is far easier to follow the straight-line form.

```
int AdamNet_ReadBlock (int dev,unsigned long block,unsigned char *buf)
{
 /* 1. Name the block. */
 if (an_send_block_num (dev,block)) return -1;

 /* 2. RECEIVE: re-poll until the device finishes seeking and ACKs. */
 gettimeofday (&t0,NULL);
 for (;;)
 {
  if (an_send_byte (CMD(MN_RECEIVE,dev))) return -1;
  r=an_recv_byte (TMO_RECV_POLL);
  if (r==RESP(NR_ACK,dev)) break;
  if (r==RESP(NR_NACK,dev)) return -1;
```

```
  if (r<0) { /* silent: still seeking. Re-poll until TMO_RECV_TOTAL. */
   if (an_ms_since(&t0) > TMO_RECV_TOTAL) return -1;
   continue;
  }
  return -1;                            /* unexpected byte               */
 }

 an_drain ();                           /* drop surplus seek-poll ACKs    */

 /* 3. CLR (CTS): pull the data packet [0xB0|dev][len BE=1024][1024][cksum]. */
 if (an_send_byte (CMD(MN_CLR,dev))) return -1;
 do { if (an_recv (hdr,1,TMO_DATA)) return -1; } while (hdr[0]==RESP(NR_ACK,dev));
 if (hdr[0]!=RESP(NR_SEND,dev)) return -1;
 if (an_recv (hdr+1,2,TMO_DATA)) return -1;
 len=(hdr[1]<<8)|hdr[2];                /* big-endian length             */
 if (len!=ADAMNET_BLOCK_SIZE) return -1;
 if (an_recv (buf,ADAMNET_BLOCK_SIZE,TMO_DATA)) return -1;
 if (an_recv_byte (TMO_DATA)<0) return -1;   /* checksum byte             */
 return 0;
}
```

Three details earn their place, and each is a Part IV pitfall in miniature:

- **The `RECEIVE` re-poll loop** (step 2) is the seek-stall tolerance. `r<0` means "no byte within `TMO_RECV_POLL` (5 ms)" — the device is still working, so loop, bounded by an overall budget `TMO_RECV_TOTAL`. Chapter 16 explains why that budget is 20 seconds.
- **`an_drain()` before `CLR`** drops ACKs the device queued in reply to our **earlier** re-polls. Chapter 13 is the whole story.
- **The `do/while` skipping a stray `NR_ACK`** before the `NR_SEND` header is belt-and-suspenders for the same desync. If a buffered ACK slips past the drain, we discard it rather than mistaking it for the data header.

8.3 Writing a block

The write is the handshake from Chapter 5′s second diagram, with the length spelled `0x04, 0x00` for 1024 — and here it is **big-endian high byte first**, matching the data-packet convention:

```
int AdamNet_WriteBlock (int dev,unsigned long block,const unsigned char *buf)
{
 if (an_send_block_num (dev,block)) return -1;
 if (an_send_byte (CMD(MN_SEND,dev))) return -1;
 if (an_send_byte (0x04)) return -1;     /* length high (0x0400 = 1024)   */
 if (an_send_byte (0x00)) return -1;     /* length low                    */
 if (an_send (buf,ADAMNET_BLOCK_SIZE)) return -1;
 if (an_send_byte (an_checksum (buf,ADAMNET_BLOCK_SIZE))) return -1;
 if (an_recv_byte (TMO_ACK)!=RESP(NR_ACK,dev)) return -1;
 return 0;
}
```

8.4 Character devices

The char transactions implement Chapter 5′s last diagram. `AdamNet_CharWrite` sends a length-prefixed payload and waits for the ACK; `AdamNet_CharRead` issues `RECEIVE` (to stage), then `CLR` (to pull), tolerating a NACK at either step as "nothing available":

```
int AdamNet_CharRead (int dev,unsigned char *buf,int maxlen,int *got)
{
```

```
 /* 1. RECEIVE: device stages pending data and ACKs, or NACKs if none. */
 if (an_send_byte (CMD(MN_RECEIVE,dev))) return -1;
 r=an_recv_byte (TMO_ACK);
 if (r==RESP(NR_NACK,dev)) { *got=0; return 0; }   /* nothing available  */
 if (r!=RESP(NR_ACK,dev)) return -1;
 /* 2. CLR: pull [0xB0|dev][len BE][reply][cksum], NACK = nothing pending. */
 if (an_send_byte (CMD(MN_CLR,dev))) return -1;
 /* ...read header, length, min(len,maxlen) bytes, drain remainder+cksum... */
}
```

The `RECEIVE`-before-`CLR` ordering is not optional, and it is the single easiest char-device mistake to make: without the `RECEIVE`, the device's `response_len` stays zero and it NACKs the `CLR`, so every read comes back empty. We flagged it in Chapter 5; it is worth the second mention because it will cost you an afternoon if you miss it.

8.5 Reset

Finally, the simplest transaction: reset. A single byte, optionally broadcast to every forwarded device:

```
void AdamNet_ResetDevice (int dev)
{
 if (dev==0xFF)
  for (d=0;d<16;++d)
   if (AdamNet_IsForwarded (d)) an_send_byte (CMD(MN_RESET,d));
 else
  an_send_byte (CMD(MN_RESET,dev));
}
```

NOTE

For your emulator. This chapter is where the ADAM specifics live, and it is the chapter you will rewrite most for another platform — because it is literally your bus's transactions. The transferable shape is: one C function per logical operation (status, read, write, control), each a straight-line sequence of `send` / `recv`-with-timeout calls that returns 0 or −1. Build the status probe first (it is the smallest), get it answering, then the others.

CHAPTER 9

Routing device I/O to FujiNet

Chapter 6 cut the seam; this chapter fills it. `UpdateFujiNet()` is the translator between the **DCB** world (what the Z80 wrote in RAM) and the **wire** world (the master transactions of Chapter 8). It reads the request out of the DCB, runs the right transaction, and writes the result — data and status — back into RAM.

9.1 The forwarding mask

Which device IDs go to FujiNet is a bitmask in `AdamNet.c`. Bit N set means "device N is forwarded":

```
static unsigned long an_forward_mask =
    (1UL << 0x02) |                                 /* printer          */
    (1UL << 0x04)|(1UL << 0x05)|(1UL << 0x06)|(1UL << 0x07) | /* disks    */
    (1UL << 0x09)|(1UL << 0x0A)|(1UL << 0x0B)|
    (1UL << 0x0C)|(1UL << 0x0D)|(1UL << 0x0E) |     /* network          */
    (1UL << 0x0F);                                  /* Fuji gateway     */
```

`AdamNet_IsForwarded()` is one shift-and-mask against it. Keyboard (`0x01`) and tape (`0x08`) are pointedly absent, so they stay with ADAMEm's local emulation — you still type on the emulator's keyboard and load local tapes while disks and the network ride the socket.

9.2 The DCB memory layout

To translate a DCB you must know its fields. `UpdateFujiNet()` reads these offsets from `RAM[DCB+n]`:

`+0`	status / command	Z80 writes the op here; master writes the result back
`+1,+2`	buffer address (LE)	where data is read into / written from
`+3,+4`	byte count (LE)	how many bytes (capped at 1024)
`+5..+8`	block number (LE32)	which 1024-byte block
`+9`	device ID high	combined with `+16` low nibble → `dev_id`
`+16`	device ID low nibble	
`+17,+18`	block size	master writes 1024 (`0x0400`) on a status query
`+20`	status flags	master sets media-present / error bits

Table 6. The DCB fields `UpdateFujiNet()` reads and writes. Offsets are relative to the DCB base; all multi-byte values are little-endian in RAM.

The command in `RAM[DCB]` is a small integer: 0 = clear, 1 = status, 2 = reset, 3 = write, 4 = read. The master writes a **result** status back into the same byte: `0x80` for success, `0x9B` for error, and — crucially for Chapter 14 — a value with bit 7 **clear** to mean "still busy, ask again."

9.3 Translating a block operation

Here is the heart of the disk path. It decodes the DCB, runs the Chapter 8 transaction, and copies bytes between the socket and ADAM RAM:

```
if (dev_id>=4 && dev_id<=7)             /* block device (disk drive) */
{
```

```
 switch (RAM[DCB])
 {
  case 1:                                 /* request status            */
   if (AdamNet_DiskStatus (dev,&st)==0)
   {
    RAM[DCB]=0x80;
    RAM[(DCB+17)&0xFFFF]=0x00;            /* block size = 1024         */
    RAM[(DCB+18)&0xFFFF]=0x04;
    RAM[(DCB+20)&0xFFFF]&=0xF0;           /* media present             */
   }
   else RAM[DCB]=0x9B;
   break;
  case 3:                                 /* write block               */
  case 4:                                 /* read block                */
   addr =RAM[(DCB+1)&0xFFFF]+RAM[(DCB+2)&0xFFFF]*256;
   count=RAM[(DCB+3)&0xFFFF]+RAM[(DCB+4)&0xFFFF]*256;
   block=(unsigned long)RAM[(DCB+5)&0xFFFF]
        +((unsigned long)RAM[(DCB+6)&0xFFFF]<<8)
        +((unsigned long)RAM[(DCB+7)&0xFFFF]<<16)
        +((unsigned long)RAM[(DCB+8)&0xFFFF]<<24);
   if (count>1024) count=1024;
   if (RAM[DCB]==4)                       /* read                      */
   {
    if (AdamNet_ReadBlock (dev,block,buf)==0)
    {
     for (i=0;i<count;++i) RAM[(addr+i)&0xFFFF]=buf[i];
     RAM[DCB]=0x80;
    }
    else { RAM[(DCB+20)&0xFFFF]|=6; RAM[DCB]=0x9B; }
   }
   else { /* write: copy RAM→buf, AdamNet_WriteBlock, set status */ }
   break;
 }
}
```

The pattern repeats for every operation: **decode the DCB, call the master, write data back to `RAM[addr]`, set `RAM[DCB]` to `0x80` or `0x9B`.** The `&0xFFFF` on every RAM index is not decoration — a DCB near the top of memory plus an offset can wrap the 16-bit address space, and the mask makes that wrap match the real Z80's behaviour.

9.4 The "mode" parameter and reads from the DCB

`UpdateDCB` / `UpdateFujiNet` are called for both Z80 **writes** to a DCB (issuing a command) and Z80 **reads** (polling the status byte). On a pure read there is nothing to do — the last result already sits in `RAM[DCB]`:

```
/* mode 0 is the Z80 reading the DCB status byte: the result of the last
   command already sits in RAM[DCB], so just leave it. */
if (!(mode&127)) return;
```

This looks like a throwaway line. It is in fact the hook on which the entire non-blocking-read design hangs (Chapter 14): because the Z80 **polls** the status byte, the emulator gets called over and over while a read is in flight, and that is exactly the rhythm a long transaction needs to make progress without freezing the CPU. Keep it in mind.

9.5 Debugging the routing

One more touch from the real code: a verbose trace, gated behind `-verbose 8`, that logs every forwarded DCB op and its result.

```
if (Verbose&8)
 fprintf (stderr,"[FujiNet] dev=0x%02X op=%u DCB=%04X\n",dev_id,RAM[DCB],DCB);
/* ...run the transaction... */
if (Verbose&8)
 fprintf (stderr,"[FujiNet] dev=0x%02X -> result=0x%02X\n",dev_id,RAM[DCB]);
```

When something does not boot, this one flag tells you instantly whether the emulator is even **reaching** FujiNet, which device it asked for, and what came back — before you reach for Wireshark on the loopback socket.

NOTE
For your emulator. The translator is the only code that knows **both** your OS's request format and your bus's wire format, so it is irreducibly specific to your platform — but its skeleton is universal: gate on forwarded-and-connected, decode the request, dispatch to the matching master transaction, copy results back into the machine's memory, set a status the OS understands. Build a verbose trace into it from the very first line; you will live in that trace.

CHAPTER 10

Startup and the boot handshake

The I/O path is built. What is left is **when** it comes alive — and getting the startup ordering right is the difference between "the FujiNet drive is there when the machine boots" and "the machine drops to its built-in word processor because no disk answered in time." This chapter is short but every line of it was learned by watching a boot fail.

10.1 The command-line option

A new option, `-fujinet [port]`, enables the bridge. The port is optional and defaults to `ADAMNET_DEFAULT_PORT` (65216). The parser only consumes the next argument if it looks like a number, so `-fujinet` alone works:

```
case 59:                                /* -fujinet [port] */
    if (N+1 < argc && argv[N+1][0]>='0' && argv[N+1][0]<='9')
        FujiNetPort = atoi(argv[++N]);
    else
        FujiNetPort = ADAMNET_DEFAULT_PORT;
    break;
```

The default 65216 is chosen so it does not collide with other platforms' Bus-over-IP defaults (the Atari NetSIO default is 9997; an Apple build's relay default is 1985). Picking a per-platform default port is a small courtesy that prevents a very confusing class of bug, as Chapter 17 will show.

10.2 Wait for the peripheral before booting

Here is the ordering subtlety. The ADAM, at power-on, scans AdamNet for a boot device. If no disk answers, it falls through to the built-in **SmartWriter** word processor. If the emulator releases the Z80 before `fujinet-pc` has connected and is answering the bus, the scan finds nothing and you are staring at SmartWriter instead of `CONFIG`.

So `main()` waits — up to 30 seconds — for a **responsive** peripheral before starting the CPU:

```
if (FujiNetPort>0)
{
 AdamNet_Init (FujiNetPort);
 if (!AdamNet_WaitForConnection (30000))
  printf ("AdamNet: no fujinet-pc connected; booting anyway "
          "(press F12 to reboot once it connects).\n");
}
StartColeco();
```

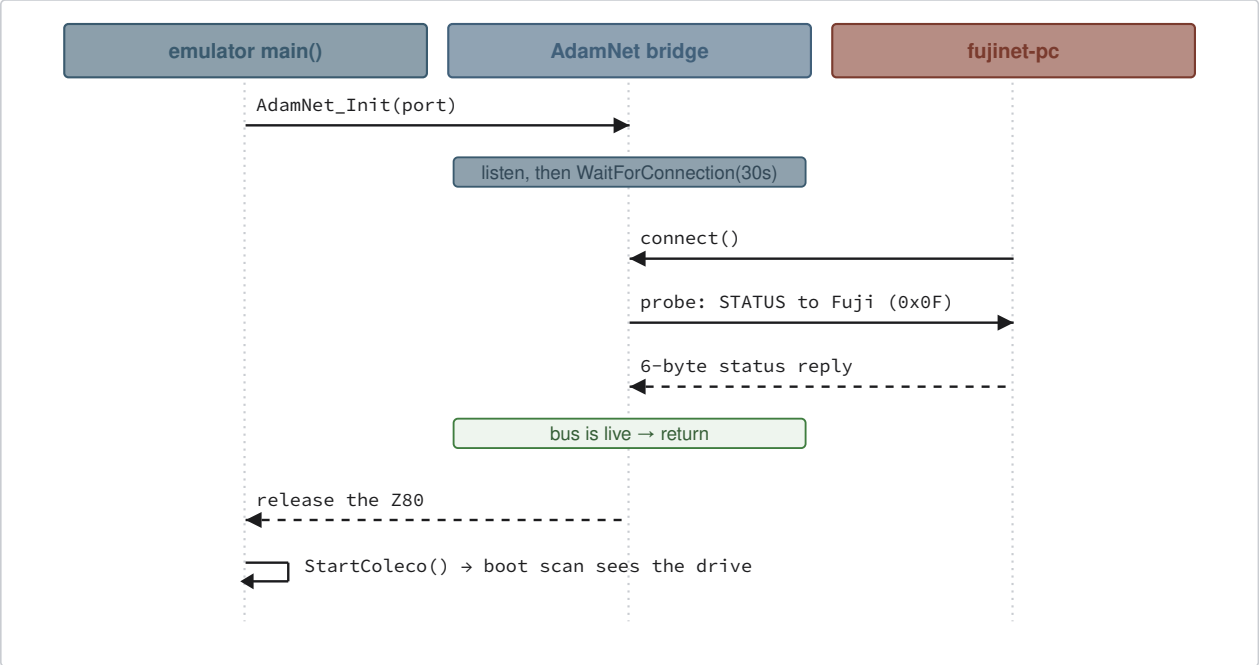

Figure 9. The boot handshake. The emulator does not just wait for a TCP connection — it waits for a peer that **answers AdamNet**, by probing the Fuji device's status, before it lets the machine boot.

10.3 Probe, don't just accept

`AdamNet_WaitForConnection()` does more than `accept()`. A bare TCP connection is not proof the peer speaks AdamNet — it might be a different platform's FujiNet that grabbed a shared default port, or a real ADAM FujiNet still finishing its own startup. So after the socket is up, the emulator **probes the bus**: it sends a STATUS to the Fuji device (`0x0F`) and waits for a well-formed reply.

```
/* socket is up; is the peer actually answering AdamNet? */
while (an_ms_since(&tp)<3000 && an_ms_since(&t0)<timeout_ms)
{
 if (AdamNet_DiskStatus (0x0F,&st)==0)
 {
  printf ("AdamNet: fujinet bus is live; booting.\n");
  return 1;
 }
 usleep (50000);
}
/* not answering within the window: probably the wrong fujinet on this port. */
printf ("AdamNet: peer isn't answering AdamNet (another fujinet on this "
        "port?); dropping it and waiting for the ADAM fujinet.\n");
if (an_conn_fd>=0) { close (an_conn_fd); an_conn_fd=-1; }
```

If the peer does not answer AdamNet within three seconds, the emulator **drops it** and keeps waiting for one that does. This is the practical defense against the shared-port problem; Chapter 17 looks at it from the firmware's side.

> **TIP**
> Using the **status probe** as the liveness check is why Chapter 5 called it "the simplest complete transaction." It needs no disk mounted, no state, and no side effects — it is the perfect heartbeat. If your bus has an equally cheap "are you there?" exchange, use it for exactly this.

> **NOTE**
> **For your emulator.** Two lessons transfer. First, **boot ordering matters**: if your machine gives up on absent boot media, wait for the peripheral before releasing the CPU. Second, **a connection is not a contract**: probe that the peer speaks your bus before trusting it, especially if your default port could be shared with another platform's bridge.

CHAPTER 11

Building and running

The last mile: compiling the bridge into the emulator and running the pair. This is mechanical, but two of the four Makefile changes are easy to miss and both produce baffling failures, so they are called out.

11.1 Makefile changes

Four edits to `Makefile.SDL` add the bridge:

1. **Compile the new translation unit.** Add `AdamNet.o` to `OBJECTS`.
2. **Declare its dependencies** so it rebuilds when the header changes: `AdamNet.o: AdamNet.c AdamNet.h`, and add `AdamNet.h` to the prerequisites of `ADAMEm.o` and `Coleco.o` (both now `#include "AdamNet.h"`).
3. **Link dynamically, not statically.** The link line changed from `sdl2-config --static-libs` to `sdl2-config --libs`. A fully static link can break the C library's name-resolution path (`getaddrinfo`), which the socket layer needs; a normal dynamic link avoids it.
4. **Enable the verbose-trace machinery** if you want the `-verbose 8` logging during bring-up (`-DPRINT_MEM -DPRINT_IO` in `CFLAGS`).

```
OBJECTS = ADAMEm.o Coleco.o Z80.o AdamemSDL.o AdamSDLSound_2.o Sound.o \
          Z80Debug.o Bitmap.o HarddiskIDE.o sms_ntsc.o AdamNet.o
adamem: $(OBJECTS)
  $(LD) -s -o adamem $(OBJECTS) -lz `sdl2-config --libs` -lm
AdamNet.o:  AdamNet.c AdamNet.h
```

PITFALL

The static-to-dynamic link change is the kind of thing that looks unrelated and gets reverted by a well-meaning cleanup. If your bridge connects but **name resolution** of a TNFS host fails only in a static build, this is why. `fujinet-pc` resolves hostnames; a statically linked `glibc` may not.

11.2 Running the pair

Order does not matter much thanks to the connect-retry and wait-for-peer logic, but the natural sequence is:

```
# 1. Start the emulator; it listens on 65216 and waits for fujinet-pc.
$ ./adamem -fujinet game.ddp

# 2. In another terminal, start fujinet-pc built for the ADAM target.
#    Its BoIP default is localhost:65216, so it connects straight in.
$ ./fujinet-pc            # (from the fujinet-pc-adam build)
```

The emulator prints its progress, and with `-verbose 8` you see every forwarded DCB op:

```
AdamNet: listening for fujinet-pc on TCP port 65216
AdamNet: waiting up to 30s for a responsive fujinet-pc ...
AdamNet: fujinet-pc connected
AdamNet: probing fujinet bus...
AdamNet: fujinet bus is live; booting.
[FujiNet] dev=0x0F op=1 DCB=FC10
[FujiNet] dev=0x0F -> result=0x80
```

That last pair of lines is the whole system working: the ADAM asked the Fuji device (`0x0F`) for status (`op=1`), and FujiNet answered success (`0x80`), over a socket, from the real firmware. Everything else in this manual is in service of making those two lines appear, and then keep appearing reliably — which is what Part IV is about.

> **NOTE**
> **For your emulator.** Building is the easy part; the only transferable warnings are real. Link the socket code the way your platform's name resolver expects (dynamic, usually), and build a verbose I/O trace you can switch on, because you will spend your debugging time reading it, not the code.

PART IV

Getting It Right

The first version of the bridge booted a local disk and looked finished. It was not. Five pitfalls stood between “it boots” and “it boots reliably over a laggy internet TNFS link without the music stuttering,” and a sixth lives on the peripheral’s side of the socket. Each was a real bug with a real commit; each teaches something that generalises. This part is why the manual exists.

CHAPTER 12

The half-duplex echo

The first pitfall is not a bug you will hit — it is a property of the bus you must **not** break, and it explains a piece of `fujinet-pc` that otherwise looks bizarre. Understand it now and two later mysteries dissolve.

12.1 One wire hears itself

AdamNet is, physically, a **single wire** shared by everyone (Chapter 5). When the master transmits a byte, every node on the wire sees it — **including the master itself**. The same is true in reverse: when a device transmits, it hears its own bytes echoed back. This is not a bug in the hardware; it is what "one shared wire" means. The firmware relies on it: after a device sends a response, it reads back and discards the echo of its own transmission to know the wire is clear again (`drain_echo()` / `wait_for_idle()` in the bus service).

12.2 TCP does not echo

A point-to-point TCP socket has no such property. When `fujinet-pc` sends bytes to the emulator, it does **not** get them back. If nothing accounted for that, the firmware's `drain_echo()` would block forever waiting for an echo that never comes — or, worse, it would consume the **master's next command**, mistaking it for the echo it expected.

12.3 fujinet-pc fakes the wire

The fix lives entirely on the firmware side, in `NetAdamNet::dataOut()`: when it transmits over the socket, it **also appends the transmitted bytes to its own receive FIFO** — a local echo that reproduces the one-wire behaviour.

```
size_t NetAdamNet::dataOut(const void *buffer, size_t length)
{
    // Local echo: reproduce the one-wire half-duplex echo so the bus
    // service's drain_echo()/wait_for_idle() consume our own transmission,
    // not the master's next command.
    _fifo.append((const char *)buffer, length);
    /* ...then actually send() it over the socket... */
}
```

Figure 10. Left: on the wire, the sender hears its own bytes for free. Right: TCP does not echo, so `fujinet-pc` appends each transmitted byte to its own RX FIFO to reproduce the effect. The bus service above it is then **unchanged**.

This is the payoff of carrying **raw bus bytes** rather than a cleaned-up protocol (Chapter 3): the firmware's bus logic can stay byte-for-byte identical to the hardware build, and the one place reality differs — the missing echo — is patched in the transport with three lines.

12.4 What the emulator must not do

For you, the emulator author, the lesson is a prohibition: **do not echo on the master side, and do not expect an echo.** The emulator's `an_recv()` reads only what the device actually sends. If you "helpfully" looped the master's

transmissions back, you would feed the firmware's echo logic twice and desync everything. The socket is point-to-point; keep it that way and let the peripheral simulate the wire's quirk, because the peripheral is the side that depends on it.

> **NOTE**
> **For your emulator.** If your bus is also a shared line (many are: the Atari SIO command/data lines, any open-collector party-line), the **peripheral** side likely depends on hearing itself. That dependence is the firmware's to satisfy, not yours — but you must know it exists, because it dictates that your transport stays a dumb point-to-point pipe.

CHAPTER 13

Slow media: the seek stall and duplicate-ACK desync

This is the first **real** bug, and the most instructive in the book, because it was invisible until the disk got slow. It booted perfectly from local storage and failed every time over TNFS. (`adamem_sdl` commit `30248a5` .)

13.1 A fast disk hides the bug

Recall the block-read handshake (Chapter 5, Chapter 8): after naming the block, the master sends `MN_RECEIVE` and **re-polls it every few milliseconds** while the device seeks, breaking out on the first `RESP.ACK` . On a local SD read the device is ready almost immediately, so the master sends one or two `RECEIVE` s and gets one ACK. Clean.

Now make the read slow — a `.ddp` image streamed from `apps.irata.online` over TNFS, across the public internet. The device stays silent for tens of milliseconds while it fetches, and the master re-polls `MN_RECEIVE` **many times**. When the block finally arrives, the device ACKs — but it ACKs **each** of those buffered re-polls it had queued up. Several `RESP.ACK` bytes are now sitting in the socket.

13.2 The desync

The master broke out of the re-poll loop on the **first** ACK, sent `CONTROL.CLR` , and then read the **next** byte expecting the data-packet header `0xB0|dev` — but got a **leftover ACK** (`0x9|dev`) instead. The read failed, the byte stream was now offset by one packet, and the error cascaded into the next block. A big `.ddp` boot crawled or stalled outright.

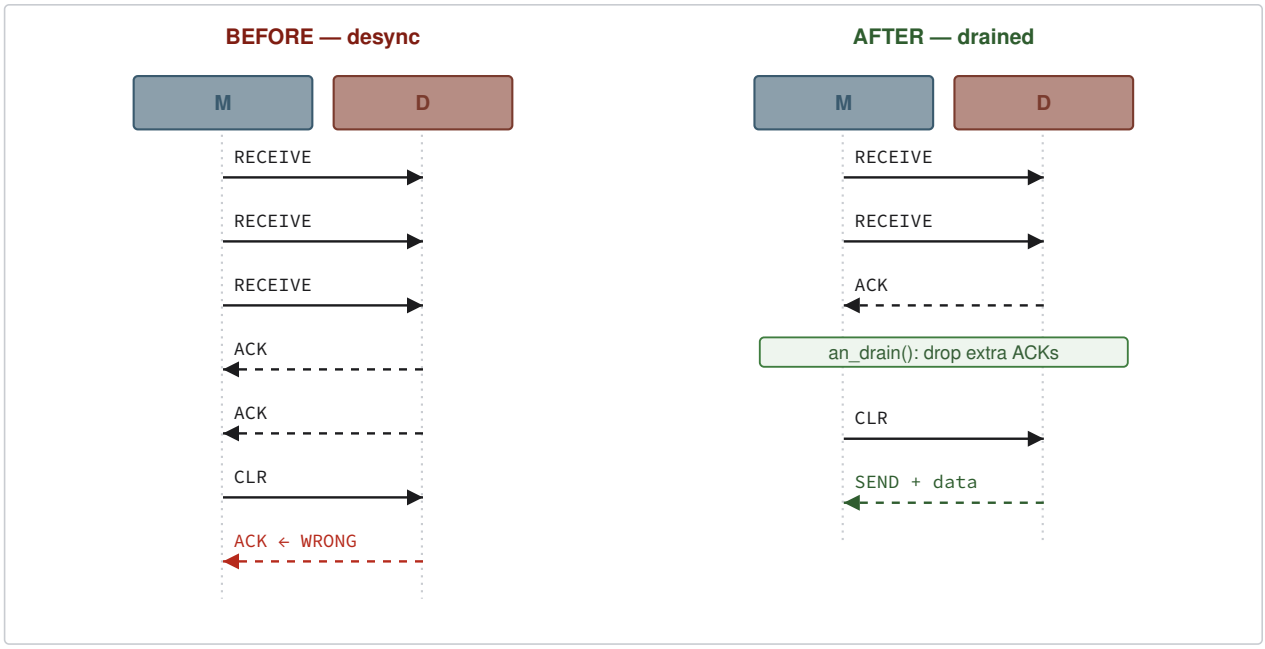

Figure 11. The duplicate-ACK desync and its fix. A slow seek leaves surplus `ACK` s queued from the master's re-polls; draining them before `CLR` realigns the stream so the data header lands where the master expects it.

13.3 The fix: drain, then forgive

Two defenses, both seen already in Chapter 8 and now explained:

1. `an_drain()` **after the re-poll loop wins its ACK, before** `CLR`**.** It non-blockingly discards the surplus ACKs the device queued. This is the primary fix.
2. **Skip a stray leading ACK** when reading the `CLR` response, in case one slips past the drain: `do { recv(hdr,1) } while (hdr[0]==RESP(NR_ACK,dev));`

The same commit also widened the data timeouts (`TMO_DATA` 1500 → 8000 ms, `TMO_RECV_TOTAL` 800 → 8000 ms) so a genuinely slow TNFS fetch is not cut off mid-read. On localhost these ceilings never fire; they exist only for the slow link. (We will widen one of them further in Chapter 16.)

> **PITFALL**
> This is the canonical "works on my machine" bug: **fast local media masks a protocol-timing flaw that only a slow remote device exposes.** If you test your bridge only against a local folder, you have not tested the re-poll path at all. Test against a deliberately slow or distant TNFS host before you believe the disk path is correct.

> **NOTE**
> **For your emulator.** Any bus with a "device, go fetch this, I will wait" step has this hazard: a slow device plus an eager re-poll equals a backlog of buffered acknowledgements. The general fix is the same — **resynchronise before the next phase**: drain what is queued, and treat an unexpected ACK where data belongs as skippable rather than fatal.

CHAPTER 14

Don’t freeze the machine: non-blocking reads

The seek-stall fix made TNFS boots **correct**. They were still **ugly**: during a multi-block load the sound stuttered and the video hitched. The cause is the deepest design lesson in Part IV. (`adamem_sdl` commit `117fc14` .)

14.1 A blocking read freezes the CPU

`AdamNet_ReadBlock()` (Chapter 8) runs the whole transaction synchronously. It is called from `UpdateFujiNet()` , which is called from the Z80′s DCB access, which is called from **the CPU emulation loop**. So while the master spins re-polling `MN_RECEIVE` for a slow block — up to tens of milliseconds — the emulated Z80 is not executing. And on the ADAM, like most of these machines, the sound and the video interrupts are driven **by the CPU**. Freeze the Z80 and the VDP interrupt stops firing, the music driver stops advancing, and the screen hitches. Block after block during a load, and you get an audible stutter.

Figure 12. Blocking vs. non-blocking reads over one slow block. Blocking holds the CPU for the whole transaction; non-blocking interleaves cheap polls with CPU execution, exactly as real hardware does.

14.2 Do it the way the hardware does

On real hardware the Z80 does not freeze during a disk read: it issues the command and **polls the drive’s status** while the drive works, running other code (the music driver) in between. The fix makes the emulator do the same, by splitting the read into a **begin** and a **poll**:

```
int AdamNet_ReadBlockBegin (int dev,unsigned long block);  /* kick it off */
int AdamNet_ReadBlockReady (int dev,unsigned char *buf);   /* 1=done 0=busy -1=err */
```

`Begin` sends the block number and the first `MN_RECEIVE` . `Ready` is a **non-blocking** poll: it peeks the socket for the ACK with a zero timeout (the special case from Chapter 7), and either finishes the transaction (`CLR` + data, returns 1), reports still-seeking (returns 0), or errors (-1).

`UpdateFujiNet()` then never blocks. When a read is in flight it leaves the DCB marked **busy** — status bit 7 clear — and services the in-flight read a little on each DCB access:

```
/* read in flight: leave DCB busy (bit 7 clear); EOS will poll again. */
r = AdamNet_ReadBlockReady (dev,buf);
if (r==1) { copy buf→RAM; RAM[DCB]=0x80; }       /* done   */
else if (r<0) { RAM[(DCB+20)]|=6; RAM[DCB]=0x9B; } /* error  */
/* else r==0: still busy, RAM[DCB] keeps bit 7 clear → EOS re-polls */
```

This is where the "EOS polls the status byte" hook from Chapter 9 pays off. EOS **already** spins reading the DCB status while waiting for a command to finish — that is how the ADAM's own drives work — so the emulator is called again and again, and each call nudges the read forward. The Z80 keeps running between calls; interrupts fire; the music plays. Only one read is in flight at a time, which is fine because EOS is sequential.

> **TIP**
> The principle is bigger than disk reads: **never run a multi-millisecond bus transaction inside the CPU loop.** Model it as a state machine the CPU polls, mirroring how the real OS waits on real hardware. If your emulated OS polls a status register or a "busy" flag, that poll **is** your service tick — hang the transaction's progress off it.

> **NOTE**
> **For your emulator.** This is the lesson most likely to bite you and least likely to show up in a quick local test. If your platform's OS issues an I/O request and then polls for completion (almost all do), make your forwarded transaction asynchronous and advance it on each poll. If instead your OS blocks on an interrupt, you will need to fire that interrupt when the transaction completes — but the same "don't stall the CPU loop" rule holds.

CHAPTER 15

Don’t storm the kernel: throttled polling

The non-blocking read fixed the stutter — and introduced a subtler one. The music played, but its **tempo** was now slightly uneven, an “odd rhythm” you could hear but not quite point at. This pitfall is a lovely example of a fix creating its own problem one layer down. (`adamem_sdl` commit `63a8b72` .)

15.1 A poll that costs a syscall

Recall **why** the non-blocking read works: EOS spins reading the DCB status byte while a command is in flight, and each read drives `UpdateFujiNet()` , which calls `AdamNet_ReadBlockReady()` . The catch is **how often** EOS spins — **thousands of times per frame**. And the first version of `ReadBlockReady()` peeked the socket on every single call:

```
r = an_recv_byte (0);    /* zero-timeout peek == a select() + recv() syscall */
```

A zero-timeout `an_recv_byte` is still a `select()` followed by a `recv()` — two system calls. Thousands of frame-spins times two syscalls each is a **syscall storm**: the kernel work alone overran the frame budget, the frame pacer skipped and caught up, and the VDP interrupts — and the interrupt-driven music — landed unevenly.

15.2 Gate the socket on a cheap clock

The data does not arrive thousands of times per frame; it arrives once, after some milliseconds. So there is no reason to **touch the socket** more than about once per millisecond. The fix gates the real socket access on `gettimeofday()` — which on Linux is a **vDSO** read, not a syscall, so it is nearly free:

```
gettimeofday (&now,NULL);
us = elapsed_since (&an_rb_poll, &now);
if (us < 1000) return 0;          /* throttle: report "pending" immediately */
an_rb_poll = now;
/* ...only now do the select()/recv() peek... */
```

The EOS status spin stays cheap (it almost always returns “pending” after a single `gettimeofday`), the frame timing stays steady, and the ACK is still picked up within about a millisecond of arriving — imperceptible to the load, inaudible to the music.

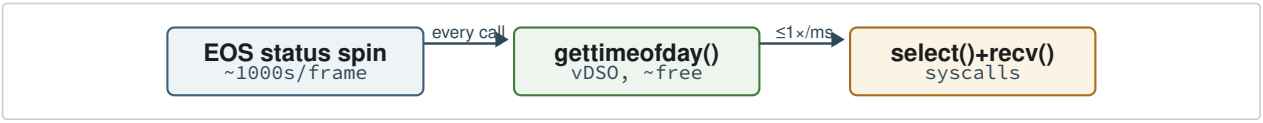

Figure 13. The throttle. A near-free clock read absorbs the thousands of EOS polls; only about one per millisecond is allowed through to the costly socket syscalls.

PITFALL

The trap is that the obvious non-blocking poll is “correct” — it never blocks, it returns promptly — and still wrong, because **correctness is not cost**. When a hot path is driven by an emulated CPU’s busy-wait, the per-call cost is multiplied by a number you did not choose. Measure how often your service tick is actually called before you put a syscall in it.

NOTE

> **For your emulator.** Whenever you hang work off an emulated busy-wait poll (Chapter 14′s technique), assume that poll fires absurdly often, and make the common "nothing to do yet" answer as cheap as a comparison. Reserve the expensive check (a syscall, a lock, an allocation) for a rate you set yourself with a cheap clock.

CHAPTER 16

Timeout budgets for lossy links

The last emulator-side pitfall is a one-line change with a paragraph of reasoning behind it, and it closes out the "make TNFS reliable" arc. (`adamem_sdl` commit `fda05e7`.)

16.1 When the device legitimately takes 16 seconds

By Chapter 13 the master waits out a slow block, bounded by `TMO_RECV_TOTAL` (then 8000 ms). That is plenty for a healthy TNFS read. But a **lossy** link is different: FujiNet's TNFS layer retries a dropped request, and in the worst case it retries up to about **eight times with a two-second timeout each** — roughly sixteen seconds before it gives up or succeeds. Booting `iss.ddp` off `apps.irata.online` over a flaky connection hit exactly this.

With an 8-second budget, the master gave up mid-read while FujiNet was still legitimately retrying. The ADAM then re-requested the block, FujiNet started over, and a 27-block boot crawled or stalled — not because anything was broken, but because the master was **less patient than the device's own retry logic**.

16.2 Be more patient than the device

The fix is to make the master's budget exceed FujiNet's worst-case retry sequence:

```
/* FujiNet may retry a lossy TNFS read up to ~8 x 2000ms, so wait longer
   than that before giving up -- otherwise the master times out mid-read,
   the ADAM retries, and a big .ddp boot over a laggy link crawls. */
#define TMO_RECV_TOTAL 20000
```

Twenty seconds. It feels enormous, and on localhost or SD it never fires — the read completes in microseconds. It exists solely so that, on a bad link, the master **waits out the device's own recovery** instead of fighting it. And it is safe: EOS keeps polling the (busy) DCB and only re-requests on the master's **error** return, so the longer wait cannot cause a premature double-request.

> **IMPORTANT**
> The rule: **the master's patience must exceed the peripheral's worst-case recovery time.** If your peripheral retries internally — and a networked one will — a too-short master timeout converts the peripheral's successful (if slow) retry into a failed transaction plus a redundant retry from the OS. Find your peripheral's worst-case and set your budget above it.

> **NOTE**
> **For your emulator.** Timeouts on the master are not "how long until I declare failure" in isolation; they interact with two other retry loops — the peripheral's internal retries and the emulated OS's command retries. Size the master's timeout so it is the **most patient** of the three for transient slowness, and let the OS's retry be the real backstop. Otherwise the three loops beat against each other and a slow link becomes a stalled one.

CHAPTER 17

The peripheral's point of view

The five pitfalls so far were all on the emulator (master) side. To finish Part IV we cross the socket and look at the changes on the **peripheral** side — in `fujinet-pc` itself. You did not write this code, and for most emulator authors it is already done. But understanding it explains the last two mysteries and tells you what to ask for if your platform's firmware is not yet PC-ready.

17.1 The 300-microsecond response deadline

Real AdamNet is a hard-real-time bus. A device must begin its response within **300 microseconds** of the command, or the master assumes it is dead. The firmware enforces this (`ADAMNET_RESPONSE_DEADLINE_US` in `adamnet.h`) by **not transmitting** a response that missed the window — on hardware, a late reply would collide with the master's next action and corrupt the bus.

Over a socket, that deadline is a problem. A PC under load, or a TNFS fetch, can easily blow 300 µs of wall-clock between reading the command and having the answer ready. If the firmware suppressed every late response, slow disk reads would never complete.

The resolution is a targeted exception. A per-device flag, `_pc_no_response_deadline` , lets a device transmit its ACK/NACK **even past** the 300 µs window — but only for devices the master **re-polls**:

```
// PC/BoIP only: when set, always transmit ACK/NACK even past the 300us
// hardware response window (a slow host can blow it). Safe only for devices
// the master re-polls (block devices); single-shot char/network devices keep
// the deadline so a late response can't pollute the next command's reply.
bool _pc_no_response_deadline = false;
```

Block (disk) devices	waived	The master re-polls `RECEIVE` ; a late ACK simply answers the next poll. Safe (commits `2530f1b54` , `a3e100420`).
Fuji + network devices	waived	Also re-polled in the BoIP master path; a slow HTTP/JSON op would otherwise miss the window (commit `78858d46a`).
Single-shot char devices	enforced	Not re-polled — a late reply would land on the **next** command's turn and corrupt it.

Table 7. Which devices waive the 300 µs deadline on the PC build, and why it is safe only for the re-polled ones.

Figure 14. The deadline waiver is per device class. The safety argument is entirely about whether the master re-polls: a re-polled device can answer late harmlessly; a single-shot device cannot.

This is the mirror image of the master's re-poll loop (Chapter 13). The master re-polls **because** the device might be slow; the device may safely answer late **because** the master re-polls. The two sides were co-designed.

17.2 The shared-port problem, from the other side

Chapter 10 had the emulator **probe** a new connection to confirm it speaks AdamNet, and drop a peer that does not. That defense exists because the BoIP default ports were, historically, shared — an Apple build and an ADAM build could both default to the same relay port and connect to the wrong emulator.

The firmware side of the fix is simply **a distinct default port per platform**, set at compile time:

```
#elif defined(BUILD_ADAM)
// AdamNet-over-IP default port (matches ADAMEm's -fujinet default)
#  define CONFIG_DEFAULT_BOIP_PORT 65216
```

Between the emulator's probe and the firmware's per-platform port, a misconnection now either does not happen or is detected and dropped. Both defenses are cheap; keeping both is belt-and-suspenders against a genuinely confusing failure mode.

17.3 Polite reconnection

One last touch worth copying. `fujinet-pc` is the connecting side, and it may run for hours as a service while the emulator is absent. So it resolves the host **once** and logs the "waiting for the emulator" notice **once** per offline period, then retries quietly:

```
// ADAMEm is often offline (it may run as a systemd service that waits for
// the emulator for hours); re-resolving and logging on every retry would
// spam the journal.
```

Combined with a reconnect throttle (one attempt per second), the result is a peripheral that waits patiently and silently for its master, reconnects automatically when the emulator restarts, and never floods a log. It is the small, unglamorous engineering that makes "leave it running" actually pleasant.

> **NOTE**
> **For your emulator.** The peripheral-side changes are the firmware team's job, but as the emulator author you should know they exist, because they define the **contract** your master relies on: late responses are tolerated for re-polled devices (so your re-poll loop is load-bearing), the port is platform-specific (so your default should match), and the peer reconnects on its own (so your listener should accept a fresh connection at any time, not just at startup).

PART V

Your Turn

The worked example is behind you. This part lifts it into a checklist you can apply to any emulator, shows how to stand up `fujinet-pc` as your reference peripheral, and collects the reference material — the wire protocol, the DCB layout, the options, a troubleshooting table, and a map of every change.

CHAPTER 18

A recipe for any emulator

The ADAM was the example, not the point. Strip the ADAM-specific names away and the same eight steps adapt almost any emulator to almost any FujiNet platform. Here they are, each pointing back at the chapter that did it for real.

18.1 The eight steps

1. **Confirm the firmware builds for PC.** `fujinet-pc` must compile for your platform's device suite. For ADAM/Atari/Apple II/CoCo/RS-232 it already does; if yours does not, that is the prerequisite (Chapter 19), and it is a firmware task, not an emulator task.
2. **Learn your bus's wire protocol.** Find the byte format, the command/response codes, the packet framing, the checksum, and the read/write/status handshakes — from the firmware's own bus code, not from folklore (Part II).
3. **Find the seam.** Locate the one function in your emulator that already receives a fully-formed peripheral request and knows where the answer goes. Add a two-predicate guard: forward only if this device is in your set **and** the peripheral is connected (Chapter 6).
4. **Open a dumb pipe.** A stream socket carrying raw bus bytes. Decide who listens (match your platform's precedent), set `TCP_NODELAY`, and write a `recv`-with-timeout plus a zero-timeout peek (Chapter 7).
5. **Build the master state machine.** One C function per logical operation (status, read, write, control), each a straight-line `send` / `recv` sequence returning 0 or −1. Build the status probe first (Chapter 8).
6. **Write the translator.** Decode the request at the seam, dispatch to the matching master transaction, copy results back into the machine's memory, set a status the OS understands — and trace every op behind a verbose flag (Chapter 9).
7. **Get the boot ordering right.** Wait for a peripheral that **answers your bus** (probe it, do not just accept the socket) before releasing the CPU, so the first boot scan sees the device (Chapter 10).
8. **Make it reliable under load.** Drain stale acknowledgements before each phase; make slow reads non-blocking and advance them on the OS's status poll; throttle that poll's socket access with a cheap clock; and set the master's timeout above the peripheral's worst-case retry (Part IV).

Figure 15. The dependency order. Each step rests on the one before; the last is where "it works" becomes "it works reliably."

18.2 A concept map across platforms

When you sit down with your own emulator, the first job is translation — not of code, but of **concepts**. This table is the Rosetta stone:

Bus master	6801 network processor → `UpdateDCB`	your I/O dispatcher
Wire protocol	AdamNet (62500-baud one-wire)	SIO / DriveWire / …
High-level request	Device Control Block in RAM	SIO block / INT regs / …
The seam	`UpdateDCB()` in `Coleco.c`	?
Liveness probe	STATUS to Fuji device `0x0F`	your cheapest exchange
Async hook	EOS polls DCB status byte	your OS's completion poll
Default BoIP port	65216	pick a distinct one

Table 8. Fill in the right-hand column for your platform before you write a line of code. If a row is blank, you have not finished Part II for your machine.

CHAPTER 19

Running fujinet-pc as your reference peripheral

You will spend the project with `fujinet-pc` running next to your emulator. This chapter is the orientation you need to drive it; it is not a substitute for the FujiNet build documentation, but it covers the ADAM specifics and the BoIP knobs.

19.1 Building the ADAM PC target

The ADAM target is built from the `fujinet-pc-adam` tree with the project's `build.sh`:

```
$ ./build.sh -p ADAM        # configure + build the ADAM PC target
```

This produces a desktop `fujinet-pc` that, unlike the ESP32 firmware, has **no real AdamNet hardware** — so for the ADAM PC build, Bus over IP is **on by default**, pointed at `localhost:65216` (`fnConfig.h`):

```
#if defined(BUILD_ADAM) && !defined(ESP_PLATFORM)
    // The ADAM PC build has no real AdamNet hardware; talk to ADAMEm over IP
    // by default (localhost:CONFIG_DEFAULT_BOIP_PORT).
    bool boip_enabled = true;
#endif
```

So in the common case you build it, run it, and it connects to your emulator with no configuration at all.

19.2 How the transport gets selected

Worth seeing once, because it is the whole firmware-side seam in one function. `systemBus::setup()` chooses the transport at startup: the real UART on the ESP32, or `NetAdamNet` (the socket) on the PC when BoIP is enabled:

```
#else  // PC build
    if (Config.get_boip_enabled())
    {
        _netadam.begin(Config.get_boip_host(), Config.get_boip_port());
        _port = &_netadam;          // bus reads/writes go to the socket
    }
    else { _serial.begin(...); _port = &_serial; }
#endif
```

Everything above `_port` — every device, every media handler, the whole command processor — is identical to the hardware build. That one pointer is where "real firmware" meets "your socket." When you read the FujiNet source, trace from `_port` outward and you are reading exactly the code your emulator will exercise.

19.3 Configuring and using it

Once the pair is connected and the ADAM has booted `CONFIG`, you use FujiNet through the emulated machine exactly as on hardware: the `CONFIG` program lists hosts and drive slots, you mount disk images (local, microSD-image, or from a TNFS host), and you reboot into them. `fujinet-pc` also exposes the same web UI as the hardware for host and slot management. Anything FujiNet can do on the ADAM, it now does in your emulator — that was the entire point.

19.4 If your platform's firmware is not PC-ready yet

For the ADAM, making the firmware run on the PC at all required some work that is invisible once done but worth naming, because your platform may need the equivalent (`fujinet-pc-adam` commit `b0e57228b`):

- **A FreeRTOS/ `esp_timer` shim** (`lib/compat/pc_rtos`) so the bus task, queues, and timers that assume an RTOS run on the PC as `std::thread` s and thread-safe FIFOs.
- **File I/O through `fnFile` / `FileHandler`** instead of raw `FILE*` , so non-stdio hosts like TNFS work (a `FILE*` -based media handler cannot read from a TNFS server). The ADAM media layer was migrated for this.
- **Shaking out latent bugs** exposed the first time the platform's code ran on a real OS with real tools — the NULL- `FILE` , char-narrowing, and response-buffer fixes mentioned in Chapter 2.

If your platform's device code has not been run on the PC before, expect a similar shakedown. It is firmware work, it is bounded, and it is the gate everything else waits behind.

> **NOTE**
> **For your emulator.** If you are on the Atari, the peripheral side is the most mature: `fujinet-pc` plus the `fujinet-emulator-bridge` NetSIO hub and Altirra is a turnkey setup. Reuse it rather than rebuilding. For other platforms, the ADAM path in this manual is the closest worked precedent.

CHAPTER 20

Closing

You now have the whole picture: the model (an emulator becomes the bus master and talks to the real firmware over a socket), the protocol (your bus's wire format, taught here as AdamNet), the seam (the one function that already brokers peripheral I/O), the state machine (one function per transaction), and the five-and-a-bit pitfalls that separate a demo from a dependable feature.

The reward is out of proportion to the work. A few hundred lines of bridge code — most of it the master state machine that is just your bus written out once — gives your emulator's users Wi-Fi, worldwide disk mounting, the genuine `CONFIG` experience, and the entire FujiNet device suite, all maintained by someone else, all guaranteed to match the hardware because it **is** the hardware's firmware. And it gives the FujiNet project the fastest development and testing loop it has.

The network really is as easy as the disk drive — once the bus says so.

APPENDIX A

AdamNet wire-protocol reference

A one-page distillation of Chapter 5 for use while coding.

Byte format. High nibble = code, low nibble = device (0–15). `CMD(c,dev) = (c<<4)|(dev&0x0F)` ; responses use the same packing.

`MN_RESET`	`0x0`	reset	`NR_STATUS`	`0x8`	status pkt
`MN_STATUS`	`0x1`	get status	`NR_ACK`	`0x9`	ack
`MN_ACK`	`0x2`	ack	`NR_CANCEL`	`0xA`	cancel
`MN_CLR`	`0x3`	CTS / send now	`NR_SEND`	`0xB`	data pkt
`MN_RECEIVE`	`0x4`	stage data	`NR_NACK`	`0xC`	nack/none
`MN_CANCEL`	`0x5`	cancel			
`MN_SEND`	`0x6`	payload follows			
`MN_NACK`	`0x7`	nack			
`MN_READY`	`0xD`	ready?			

Table 9. AdamNet command and response codes (high nibble).

Packet. `[code|dev] [len16] [payload…] [checksum]` . Checksum = XOR of all payload bytes. Block size = 1024. Status reply is fixed 6 bytes: `[0x8|dev] [len lo] [len hi] [devtype] [status] [cksum]` ; `devtype` 0x01 = block, 0x00 = char.

Block read. `SEND blk#(5,LE32+0)` → `ACK` ; then `RECEIVE` (re-poll until `ACK`); then `CLR` → `SEND` + `len(BE=1024)` +1024 bytes+ `cksum` . **Block write.** `SEND blk#` → `ACK` ; then `SEND` + `len(BE=1024)` +1024 bytes+ `cksum` → `ACK` . **Char read.** `RECEIVE` → `ACK` / `NACK` ; then `CLR` → `SEND` + `len(BE)` +data+ `cksum` .

APPENDIX B

Device Control Block reference

DCB fields read/written by `UpdateFujiNet()`, relative to the DCB base (`RAM[DCB+n]`). Multi-byte values little-endian unless noted.

`+0`	Command in (0 clear · 1 status · 2 reset · 3 write · 4 read); result out (`0x80` ok, `0x9B` err, bit 7 clear = busy)
`+1,+2`	Buffer address (LE)
`+3,+4`	Byte count (LE), capped at 1024
`+5,+6,+7,+8`	Block number (LE32)
`+9`	Device ID high nibble source
`+16`	Device ID low nibble source; `dev_id = RAM[+9]<<4 | (RAM[+16]&0x0F)`
`+17,+18`	Block size (master writes `0x0400` = 1024 on status)
`+20`	Status flags (media present / error bits)

Table 10. DCB layout as used by the ADAM bridge.

APPENDIX C

ADAMEm command-line reference

`-fujinet [port]`	`-fn`	Enable the AdamNet-over-IP bridge; listen on `[port]` (default 65216) and wait up to 30 s for a responsive peer before booting
`-verbose 8`		Log each forwarded DCB op and its result to `stderr` (`[FujiNet] dev=… op=… → result=…`)

Table 11. The options added or used by the bridge. `-fujinet` with no port uses `ADAMNET_DEFAULT_PORT` ; only a numeric next argument is consumed as the port.

`AdamNet.c`	
`ADAMNET_DEFAULT_PORT`	65216 — matches `CONFIG_DEFAULT_BOIP_PORT`
`TMO_ACK`	300 ms — wait for an ACK / status reply
`TMO_DATA`	8000 ms — wait for data-packet bytes
`TMO_RECV_POLL`	5 ms — per `RECEIVE` re-poll while seeking
`TMO_RECV_TOTAL`	20000 ms — total budget to ride out a TNFS-retry seek
`ADAMNET_BLOCK_SIZE`	1024
`an_forward_mask`	bitmask of forwarded device IDs (printer/disks/network/Fuji)

Table 12. The master's tunables and their reasons (Part IV).

APPENDIX D

Troubleshooting

Boots to SmartWriter, not `CONFIG`	Emulator released the Z80 before the peer answered. Confirm `fujinet-pc` is running and BoIP is on; the emulator waits 30 s then boots anyway — press F12 to reboot once connected (Ch. 10).
Connects but every op fails	Probe failing or wrong peer on the port. Check `-verbose 8` for `[FujiNet]` lines; confirm both ends use the same port and the peer is the **ADAM** build (Ch. 10, 17).
Local disks boot, TNFS disks stall	Duplicate-ACK desync and/or too-short timeouts — the classic fast-media-masks-the-bug case. Ensure `an_drain()` runs before `CLR` and `TMO_RECV_TOTAL` is 20 s (Ch. 13, 16).
Audio/video stutters during loads	Blocking read freezing the CPU. Use the non-blocking `ReadBlockBegin` / `Ready` path and leave the DCB busy (Ch. 14).
Music tempo is subtly uneven	Socket syscall storm from the EOS status spin. Throttle the socket peek behind `gettimeofday` (Ch. 15).
Handshake is mysteriously slow (40 ms/op)	`TCP_NODELAY` not set on one end. Set it on both (Ch. 7).
TNFS hostname won't resolve	Statically linked emulator. Link the socket build dynamically (`sdl2-config --libs` , Ch. 11).
Char/network device reads come back empty	Missing `RECEIVE` before `CLR` ; the device never staged its response (Ch. 5, 8).

Table 13. Symptom-to-cure table, each pointing at the chapter with the full story.

APPENDIX E

Source map

Where each change lives, for reading the real diffs.

`adamem_sdl/AdamNet.c` `.h`	The TCP transport + the AdamNet master state machine (transport, transactions, probe, async read, drain).
`adamem_sdl/Coleco.c`	`UpdateDCB()` intercept + `UpdateFujiNet()` translator + forwarding logic + verbose trace.
`adamem_sdl/ADAMEm.c`	`-fujinet [port]` option, init, wait-for-peer in `main()`.
`adamem_sdl/Makefile.SDL`	`AdamNet.o`, dependencies, dynamic link.
commit `d0f79b0`	The bridge: transport, master, routing, option.
commit `30248a5`	Seek-stall duplicate-ACK fix (`an_drain` , timeouts).
commit `117fc14`	Non-blocking reads (`ReadBlockBegin` / `Ready`).
commit `63a8b72`	Throttled async-read polling.
commit `fda05e7`	Widened `TMO_RECV_TOTAL` for TNFS retries.
`fujinet-pc-adam/lib/hardware/NetAdamNet.*`	The socket IOChannel + local echo (peripheral side).
`fujinet-pc-adam/lib/bus/adamnet/*`	Transport selection, response deadline, idle handling.
commit `b0e57228b`	The ADAM PC target + NetAdamNet + RTOS/fnFile shims.

Table 14. A map from concept to source for both repositories.

APPENDIX F

Glossary

AdamNet the Coleco ADAM's 62500-baud half-duplex one-wire peripheral bus.

Bus over IP (BoIP) carrying a machine's peripheral-bus bytes over a network socket between an emulator and `fujinet-pc`.

DCB (Device Control Block) the in-RAM structure through which ADAM software asks the 6801 network master to perform a device operation.

EOS the ADAM's Elementary Operating System; it polls the DCB status byte while a command is in flight.

fujinet-pc the FujiNet firmware compiled to run as a desktop application.

Local echo `NetAdamNet` appending its own transmitted bytes to its receive FIFO to reproduce the one-wire bus's self-hearing.

Master the bus controller; in BoIP, the emulator.

NetSIO the Atari precedent for Bus over IP (SIO over a UDP hub).

Peripheral a device on the bus; in BoIP, `fujinet-pc`.

Re-poll the master re-issuing `MN_RECEIVE` while a device is still seeking.

TNFS the network filesystem FujiNet mounts disk images from.

Connecting an Emulator to FujiNet-PC — Revision 1, June 2026.
Built with Typst from sources in `adamem_sdl`, `fujinet-pc-adam`, and `fujinet-firmware`.
The network is as easy as the disk drive — once the bus says so.
