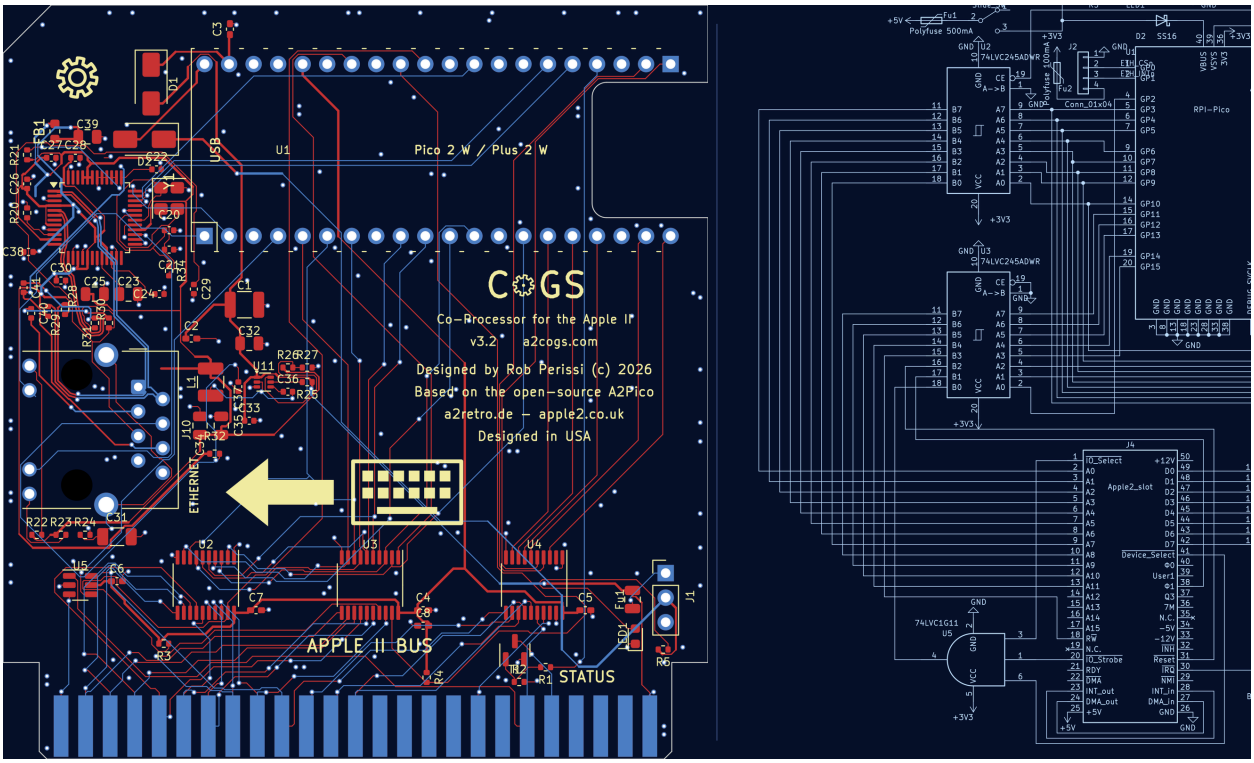
COGS

CoGS Technical Reference

Architecture, registers, protocol, firmware, and hardware of the CoGS card



Describes firmware 1.31 (9/29/2026). Manual revision 18, built 2026-10-08.
a2cogs.com

Copyright 2026 Rob Perissi. Apple, Apple II, Apple IIGS, GS/OS, ProDOS, and Finder are trademarks of Apple Inc. Raspberry Pi is a trademark of Raspberry Pi Ltd.

Contents

About This Book ... 1
1 Architecture ... 3
1.1 Two processors, two jobs ... 3
1.2 Data paths ... 3
1.3 Memory ... 3
1.4 What the host sees ... 4
2 The Bus Interface ... 5
2.1 Address regions ... 5
2.2 Register map ... 5
2.3 Presence probe ... 7
2.4 Reading the FIFO ... 7
2.5 Rules for host code ... 8
3 Framing and the Command Protocol ... 9
3.1 The frame ... 9
3.2 Opcode spaces ... 9
3.3 Payload limits ... 10
3.4 Command sequence ... 10
3.5 Errors and resynchronization ... 11
3.6 Deferred replies and polling ... 11
3.7 Capability discovery ... 11
4 Command Reference ... 13
4.1 Summary ... 13
4.2 System ... 14
4.3 Network ... 16
4.4 Network tools ... 18
4.5 Sockets ... 19
4.6 HTTP ... 20
4.7 Card services ... 21
4.8 Link layer ... 23
4.9 Network disk ... 23
4.10 Catalog sources ... 26
4.11 Byte streams ... 26
4.12 Player ... 28
4.13 Config terminal ... 28
5 Event Reference ... 30
5.1 Summary ... 30
5.2 CONNECTED (`$90`) ... 30
5.3 DATA (`$91`) ... 30
5.4 CLOSED (`$92`) ... 31
5.5 ERROR (`$93`) ... 31
5.6 WIFI_STATE (`$94`) ... 31
5.7 HTTP_DONE (`$95`) ... 31
5.8 LINK_FRAME (`$96`) ... 31
5.9 NONCE_PROGRESS (`$97`) ... 32
5.10 SEND_ACK (`$9A`) ... 32
5.11 Demultiplexing ... 32
6 The Network Disk ... 33
6.1 The slot ROM ... 33

6.2 Boot sequence ... 34
6.3 Block fetch and caching ... 35
6.4 The mount table and units ... 36
6.5 The recent-disk list ... 36
6.6 Writable volumes ... 38
6.7 Disk image formats and sources ... 38
7 Desk Accessory and Tool Injection ... 39
7.1 The `$C800` views ... 39
7.2 The config terminal ... 39
7.3 The desk accessory installer ... 40
7.4 Card-served tool sets ... 41
7.5 Residency and precedence ... 42
8 Bus Timing Rules ... 43
8.1 The read deadline ... 43
8.2 Indexed-store dead cycles ... 43
8.3 The 16-bit index extra cycle ... 44
8.4 Accelerators and re-issued I/O cycles ... 44
8.5 Interrupts during a drain ... 45
8.6 Slot RESET ... 45
9 Programming the Card ... 47
9.1 Detecting the card ... 47
9.2 Sending a command and reading the reply ... 48
9.3 Draining bulk data ... 49
9.4 HTTPS in three commands ... 50
9.5 Getting one value out of JSON ... 50
9.6 Hashing and random numbers ... 50
9.7 Mounting a disk image from a program ... 51
9.8 An 8-bit host in cc65 ... 51
9.9 The `cogslib` library ... 52
10 Hardware ... 53
10.1 Block diagram ... 53
10.2 Bus interface ... 53
10.3 Ethernet ... 54
10.4 Power ... 55
10.5 Mechanical ... 55
10.6 Board revisions ... 55
11 The Emulator ... 57
11.1 What the model implements ... 57
11.2 Differences from the card ... 58
11.3 Installing in Ample ... 59
11.4 Running headless ... 59
11.5 Scripting with Lua ... 60
11.6 Building ... 60
Appendix A. Error Codes ... 61
Appendix B. URLs, File Types, and Catalog Formats ... 63
Appendix C. Glossary ... 68
Appendix D. Credits and Licenses ... 72

About This Book

This book is the technical reference for the CoGS card. It describes the card at the level a programmer or hardware engineer needs: the bus interface and register map, the frame protocol and every command and event, the firmware's disk streaming engine, the services the card injects into a running Apple IIGS, the timing rules host code must follow, the hardware, and the emulator used to develop against the card without one in a slot.

It describes firmware 1.31 (9/29/2026). Where a feature requires a minimum firmware version, the text says so. Behavior not described here is not part of the interface and may change.

The *CoGS Owner's Manual* covers installing and using the card and is the right book for everyone else.

Organization

Chapters 1 to 3 describe the card from the outside: architecture, registers, framing. Chapters 4 and 5 are the command and event reference. Chapters 6 to 8 describe how the firmware implements disks, desk accessories, and tools, and the timing rules that follow from the bus. Chapter 9 is a programming guide with examples in 65816 assembly, ORCA/C, and cc65. Chapter 10 is the hardware. Chapter 11 is the emulator. Appendices A to C are lookup tables; Appendix D lists credits and licenses.

Acknowledgments

The card's bus interface is the A2Pico design: the firmware library by Oliver Schmidt (MIT license) and the hardware by Ralle Palaveev. CoGS reproduces that interface and links the A2Pico PIO bus library unchanged; the register semantics, timing rules, and `$C800` sharing described in this book were worked out on that foundation, and Oliver Schmidt reviewed the bus timing and the `$C800` sharing arrangement.

FujiNet demonstrated TLS, HTTPS, and JSON offload for 8-bit machines years before this card, and its JSON device model is the direct ancestor of JSON_GET. Marinetti, ORCA/C, and Golden Gate keep the IIGS a programmable target in 2026 and are the toolchain behind every GS-side example here.

The PCM player tool set the card serves as Tool 226 derives from Petar Puskarich's Tool225 Ensoniq streamer, itself built on the NinjaTracker Plus player by Ninjaforce, toolified by Brutal Deluxe Software, on work by FTA and ESP. It is used with his permission. Tool 219, the SoundSmith player, is Brutal Deluxe freeware by Antoine Vignau and Olivier Zardini, continuing the FTA tool set on Huibert Aalbers' player, and is served unmodified. Antoine Vignau's published source trees were the reference for Super Hi-Res, SoundSmith, and GS/OS behavior throughout.

The a2cogs disk catalog is built on the curated IIGS collection of What is the Apple IIGS? (whatisthe2gs.com) and on 4am's Total Replay, which the card streams from its own release. Appendix D lists the third-party components in the firmware and their licenses.

Conventions

Hexadecimal numbers are written with a leading dollar sign, `$C0A0`. Register names are set in small capitals in prose and as code in tables: DATA, STATUS. Multi-byte integers on the wire are little-endian

unless stated. Command opcodes are given as `NAME ($op)` with the response opcode in parentheses where it differs: `HASH ($30, reply $B0)`.

Code examples are 65816 assembly in Merlin and ca65 syntax, ORCA/C, or cc65 C, as marked.

Revision history

Table 1. Book revisions

Revision	Firmware	Change
1	1.03	First draft chapters.
2	1.10	Architecture chapter.
5	1.14	Complete first edition: bus interface, protocol, command and event reference, network disk and the recent-disk list, injected services, timing rules, programming guide, hardware, emulator, appendices.
7	1.14	Recent-disk list: CGR2 flags (boot, TR, your-source), Browse X and A wait if the image is still mounted, your-source host gate.
8	1.20	Shipping edition. Stream ring reserved at boot, Support Snapshot vitals, W5500 PHY autoneg and 24 MHz SPI, drive count by operating system.
9	1.21	Card ID (User-Agent `id/` tag, CARD_INFO unchanged), volume read-ahead windows capped at two.
10	1.25	Pre-production edition. VOL_EJECT and DIB type `$06`, fixed buffer slots for windows, prefetch, decoder and the CDA frame, CA roots served from flash, idle keep-alive reaper.
18	1.31	Production edition. Two-phase remount hold, share write-protect poll (`X-CoGS-Writable`, `X-CoGS-Poll`) and share-gone dismount, read-only units report `$F4`. Recent tag byte (TR, MC, HD). Link layer poll mode (`LINK_OPEN` mode bit 7, `LINK_POLL $44`, LINKQ at register +4), wired bridge with the GS's own MAC and the mirror filter, NAT datagram and checksum fixes, W5500 SHAR re-program. Pending eject (VOL_EJECT settles on the next host call, lifted by a boot) and the slot ROM's SP_SCAN boot marker from the SmartPort unit-0 STATUS, then (1.31) from every per-unit STATUS, so a GS/OS boot clears the CDA latch and ProDOS 8′s does not; ROM body 2036 bytes.

CHAPTER 1

Architecture

The CoGS card is a Raspberry Pi Pico 2 W module and a bus interface on one board. Every function of the card, from answering a slot read within 500 ns to terminating a TLS session, runs on the module's RP2350 microcontroller. This chapter describes how that work is divided and how data moves between the Apple II and the network.

1.1 Two processors, two jobs

The RP2350 has two Cortex-M33 cores at 200 MHz. They do not share work.

Core 1 owns the Apple II bus. PIO state machines capture address and data on `/DEVSEL`, `/IOSEL`, and `/IOSTRB`. Core 1 serves register reads from bytes that are already staged, queues writes into a ring, and serves the `$Cn00` and `$C800` ROM windows from RAM copies. A 6502 or 65816 read cycle must be answered within about 500 ns of ENBL falling. Nothing on core 1 blocks, calls into flash, or waits on the network.

Core 0 owns everything else. It parses frames from the host-to-card ring, dispatches commands, runs lwIP and mbedTLS, performs HTTP range GETs, maintains the disk cache, transcodes and streams audio, serves the CDA and tool blobs, and writes settings. A long TLS handshake or a flash program on core 0 must not be on the path that answers the next slot read.

Slot RESET is a GPIO interrupt on core 1. Flash erase and program on core 0 take XIP off. The firmware defers a RESET edge that lands in that window and replays it when flash is done, so core 1 does not vector into unmapped flash.

1.2 Data paths

The host sees sixteen registers at `$C0n0` through `$C0nF` (n is 8 plus the slot). Writes to DATA push the host-to-card ring. Reads from DATA or from the wide window at +B/+C pop the card-to-host ring. STATUS, RXLO/RXHI, and BULK are side-effect-free views of that ring. CONTROL pulses reset, flush, and `$C800` view changes. Chapter 2 is the map.

Disk blocks take a longer path. A SmartPort READ becomes a frame. A miss fetches a window of blocks over HTTPS (or HTTP on a LAN share), lands them in an SRAM window and in the flash warm cache, then either pushes a 9-byte header plus 512 data bytes through the FIFO or maps the 512 bytes into a `$C800` view for an MVN. A hit skips the network. The slot ROM drains with BULK plus DATA, or with the block window.

Audio and pictures ride the same rings from **CoGS Config**. Tool 226 and Tool 219 are blobs the card injects into GS/OS. They are not files on the boot volume.

1.3 Memory

The module has 520 KB SRAM and 4 MB flash.

Flash holds the firmware image, wear-leveled settings (Wi-Fi, mounts, sources, clock prefs), and the disk warm cache. The cache is 765 lines of 7 blocks in 4 KB sectors, about 2.6 MiB. The key includes the origin validator (ETag or Last-Modified when the server sends one, otherwise URL plus size). A same-size replace of a file with no validator can serve stale lines until the cache is wiped.

SRAM holds the two bus rings (4 KB host-to-card, 16 KB card-to-host), the four `$C800` views (driver body, CDA installer, config terminal, block window), TLS and HTTP buffers, and the rest of core 0′s heap.

1.4 What the host sees

A slot device. Sixteen I/O registers, a SmartPort / ProDOS ROM, and a `$C800` window the card claims and releases the usual way (`$CFFF` releases, an IOSEL cycle claims). No host program is required to boot a disk or open the card menu.

Optional GS/OS pieces are files on a boot volume: the CoGS CDev, the Status NDA, and the Marinetti link layer. **CoGS Config** is not a file. The card installs it after the Finder is up.

Published clients (Marinetti link layer, IP65, cogslib) talk the frame protocol through the registers. They do not jump into slot ROM. Presence is ID `$A2`, VERSION not `$00`, STATUS not `$FF`. VERSION `$04` means the wide pop window at +B/+C exists.

CHAPTER 2

The Bus Interface

The card occupies one Apple II slot and responds in the three address regions the slot decodes. All host software talks to the card through sixteen I/O registers in the first of them. This chapter defines those registers bit by bit, the STATUS flags, the presence probe, and the rules a host must follow when reading them.

2.1 Address regions

The slot hardware decodes three regions for the card, where *n* is the slot number.

Table 2-1. Address regions

Region	Addresses	Select line	Use
I/O registers	`$C080 + n*16` to `$C08F + n*16`	/DEVSEL	Sixteen registers; the whole protocol
Slot ROM	`$Cn00` to `$CnFF`	/IOSEL	256-byte boot and driver ROM
Shared ROM	`$C800` to `$CFFF`	/IOSTRB	2 KB driver body; the installer overlay (see Chapter 7)

Slot I/O cycles run at 1 MHz on every Apple II, including an accelerated IIGS, so register timing is identical on every host. The card answers a read within the bus deadline of about 500 ns from the fall of the select line. To meet that deadline every readable byte is staged before the host asks for it, which is why no register read has a slow side effect and why STATUS carries only flags that cost nothing to compute.

In the examples that follow, `BASE` is `$C080 + n*16`. In 65816 code the natural form is `lda BASE+off,x` with X holding `n*16`, or a long absolute `lda $E0C080+n*16+off`.

2.2 Register map

Table 2-2. I/O registers, BASE + offset

Offset	Name	Access	Function
+0	DATA	R/W	FIFO window. A write pushes one byte to the card. A read pops one byte from the card. A read with nothing queued returns `$00`; check STATUS or RXLO first.
+1	STATUS	R	Flag byte, see STATUS below. No side effects.
+2	RXLO	R	Low byte of the count of bytes queued for the host. Reading RXLO latches RXHI.
+3	RXHI	R	High byte of the count, as latched by the last RXLO read.

Offset	Name	Access	Function
+4	CONTROL / LINKQ	W / R	Write: pulse bits, see CONTROL below. Read (firmware 1.26): LINKQ, the number of inbound link frames queued for LINK_POLL, 0 to 255 saturating, 0 unless a poll-mode link is open. A pure load with no side effect; the link layer reads it from its Marinetti poll entry before issuing LINK_POLL. Older firmware returns open bus here.
+5	ID	R	Always `$A2`.
+6	VERSION	R	Register-map generation. Reads `$04`.
+7	BULK	R	`$FF` when 512 or more bytes are queued for the host, otherwise `$00`.
+8	CDA	R/W	Session flag used by the IIGS desk-accessory installer: `$A5` once the desk accessory has been queued this session. Write `$A5` to set, any other value to clear. Cleared by slot reset, by soft reset, and (firmware 1.31) when SP_SCAN shows GS/OS generating its drivers, a boot with no RESET on the bus.
+9	VOLIO	R	Wrapping 8-bit count of disk data operations (reads and writes, including cache hits).
+A	SYNC	R/W	Read: parser synchronization generation, incremented each time a flush-TX completes. Write: installer stage trace byte, diagnostic only.
+B	DATAW_L	R	Wide pop window, low half. Each read pops one byte from the same stream as DATA.
+C	DATAW_H	R	Wide pop window, high half. Same semantics as +B.
+D	USTAT_H	R/W	Unit status view, see below. Write: select a mount handle 0 to 7. Read: block count, low byte.
+E	USTAT_M	R/W	Read: block count, middle byte. Write: acknowledge the selected unit's disk-switched flag.
+F	USTAT_F	R	Unit status flags for the selected handle: bit 7 valid, bit 2 block count bit 16, bit 1 disk-switched pending, bit 0 read-only.

2.2.1 Unit status view

The three registers at +D, +E, and +F let a block driver answer a SmartPort STATUS call without a frame round trip. Write the mount handle to +D, then read +F. If bit 7 is set the card has that mount open with a final block count: +D and +E read the low and middle bytes of the count, bit 2 of +F is bit 16, bit 0 reports the read-only latch, and bit 1 reports a pending disk-switched flag, which a write to +E clears. If bit 7 is clear (not mounted, mount in progress, link not up, or the unit is under the two-phase remount hold of Chapter 6) the driver falls back to VOL_STAT and VOL_SWITCHED. The card drops bit 7 before it rewrites a changed count, so a driver that re-reads +F after the two count bytes and sees it moved knows to retry. No read of these registers has a side effect.

2.2.2 STATUS

Table 2-3. STATUS bits

Bit	Name	Meaning
7	RX_AVAIL	At least one byte is queued for the host at DATA. `lda STATUS` then `bmi` tests it.
6	TX_READY	At least one byte of space is free in the host-to-card FIFO. `bit STATUS` then `bvs` tests it.
5		Reserved, reads 0.
4	LINK_UP	The card has a network connection and an address.
3	ERROR	Sticky error latch. Cleared when the host reads the pending ERROR event.
2 to 0		Reserved, read 0.

2.2.3 CONTROL

Each bit is a pulse. Write a byte with the bit set; the action runs once.

Table 2-4. CONTROL bits

Bit	Action
0	Soft reset. Aborts every command in flight, flushes both FIFOs, drops all sockets. Settings in flash are untouched.
1	Enable the slot interrupt line. Reserved for interrupt-driven receive; the current firmware does not assert /IRQ.
2	Flush the card-to-host FIFO.
3	Flush the host-to-card FIFO and reset the frame parser. SYNC increments when the flush completes.
4	Map the config-terminal overlay over the card's `$C800` to `$CFFF` window (Chapter 7).
5	Map the desk-accessory installer over `$C800` to `$CFFF` (Chapter 7).
6	Restore the driver body to `$C800` to `$CFFF`.
7	Map the block window over `$C800` to `$C9FF` (Chapter 6).

Bits 4, 5, 6, and 7 switch only the content the card serves in the shared ROM window. The card's claim on the window follows the normal slot rule: /IOSEL asserts it, a read of `$CFFF` releases it.

2.3 Presence probe

Host software detects the card with three reads and no timing dependence:

1. ID reads `$A2`.
2. VERSION is not `$00`.
3. STATUS is not `$FF`, which is what a floating bus returns from an empty slot.

VERSION identifies the register map. Software that uses only DATA, STATUS, RXLO, RXHI, and CONTROL works with any value. Software that uses BULK, CDA, VOLIO, SYNC, or the wide pop window requires VERSION `$04` or greater; on an older card those offsets read `$00`.

2.4 Reading the FIFO

Three drain patterns cover every need. Choose by how much you are owed.

2.4.1 Byte at a time

Poll STATUS bit 7, then read DATA. Correct on every host, about 22 to 25 KB/s on a stock IIGS with the loop overhead included.

```
@wait:  lda     STATUS,x
        bpl     @wait
        lda     DATA,x
```

2.4.2 Burst drain

When a frame header has been consumed and the remaining payload is 512 bytes or more, read BULK. A nonzero BULK guarantees that at least 512 bytes are already queued, so a tight loop may pop DATA 512 times with no poll between pops. This is the pattern the disk driver uses for a 512-byte block.

2.4.3 Wide drain

Requires VERSION `$04`. On a 65816 with a 16-bit accumulator, one `lda DATAW_L,x` reads +B then +C in consecutive bus cycles and pops two bytes in wire order, low byte first.

1. Read RXLO then RXHI. The pair is atomic because RXLO latches RXHI. Clamp the count to the bytes your frame still owes you; the count includes anything the card queued behind it.
2. Pop `count & ~1` bytes as 16-bit reads at +B. No STATUS reads in the loop.
3. Pop an odd trailing byte through the STATUS-gated DATA path, and do any blocking wait there too.
4. Repeat until the frame is drained.

The wide window has a re-issue guard. A legitimate wide drain alternates +B, +C, +B, +C. If the card sees two consecutive pops at the same offset it serves the same byte again without popping. An accelerator that re-runs a stretched I/O cycle therefore cannot pop twice. Because of the guard, never pop single odd bytes through +B or +C; use DATA.

Measured on a stock 2.8 MHz IIGS, the wide drain moves about 80 KB/s including caller overhead.

2.5 Rules for host code

These rules follow from how 6502-family processors sequence bus cycles. Code that breaks them pops or pushes the FIFO more often than it intends, and the failure looks like corrupt data rather than an error.

Indexed stores to DATA are compensated; do not rely on the compensation for anything else. `sta DATA,x` and `sta DATA,y` perform a read of the target address in the cycle before the write on every 6502-family CPU, and on DATA that read would pop a byte. The card recognizes the pattern (a DATA write within 2 µs of a DATA read) and puts the byte back, so `sta DATA,x` is a correct way to write a command byte and is what the slot ROM uses. The compensation covers exactly one dummy read per store. Read-modify-write instructions on DATA (`inc DATA,x`) read twice and are not covered; host code that legitimately reads DATA and then writes it inside 2 µs cannot exist at the 1 MHz bus rate, so the compensation is never wrongly applied. DATAW_L and DATAW_H ignore writes, so an indexed store that spills onto them is harmless.

Never read a register with `abs,X` or `abs,Y` while the index register is 16 bits wide. With a 16-bit index the 65816 adds a cycle that reads the un-carried address (high byte of the base, low byte of base plus index low). When the index does not cross a page, that address is the register itself, so `lda DATA,x` pops twice and keeps the second byte. Set the index to 8 bits (`sep #$10` or `sep #$30`) around FIFO access, or use direct-page indirect long addressing (`lda [dp]`), which has no such cycle. Emulation-mode code and 8-bit hosts are unaffected. Some accelerator cores do not emit the extra cycle, so code with this fault can appear to work on one machine and fail on another.

Do not rely on empty reads. DATA returns `$00` when nothing is queued. A byte of `$00` in the stream is indistinguishable from an empty pop, so gate every pop on STATUS, BULK, or the RXLO/RXHI count.

Finish a frame before starting another. The card never interleaves bytes of two card-to-host frames, and it expects the same of the host: write a complete command frame before writing the next.

CHAPTER 3

Framing and the Command Protocol

Every exchange with the card is a frame written to or read from DATA. This chapter defines the frame, the opcode spaces, payload limits, the sequence of a command from the host's first write to the last byte of the reply, and how the two sides recover when a frame is malformed.

3.1 The frame

```
[OPCODE:1] [LEN_LO:1] [LEN_HI:1] [PAYLOAD:LEN]
```

LEN is the payload length in bytes, little-endian, and counts the payload only. A frame with no payload is three bytes. All multi-byte integers inside payloads are little-endian unless an entry in Chapter 4 says otherwise (the SHA-256 digest and the MAC address are the exceptions).

The protocol is plain byte I/O. Nothing in it depends on the width of the host's registers or on the 65816; the same frames are written from 6502 assembly, cc65 C, and ORCA/C.

3.2 Opcode spaces

Table 3-1. Opcode spaces

Range	Direction	Meaning
`$00` to `$7F`	Host to card	Commands.
`$80` OR op	Card to host	The reply to command op, formed by setting bit 7. PING `$00` replies `$80`; HASH `$30` replies `$B0`; VOL_READ `$51` replies `$D1`; STREAM_OPEN `$70` replies `$F0`.
`$90` to `$9F`	Card to host	Unsolicited events (Chapter 5).

The reply space overlaps the event space for commands `$10` to `$1F`. CONNECT `$10` completes with the CONNECTED event `$90` and CLOSE `$12` with CLOSED `$92`; those events are the replies. The SEND acknowledgement is SEND_ACK `$9A`, not `$91`, because `$91` is the DATA event.

3.3 Payload limits

Table 3-2. Payload limits

Direction	Limit	Notes
Host to card	2048 bytes	The parser rejects a longer header (see Errors below).
Card to host, DATA event	1024 bytes	So a 64 KB host can use a small fixed buffer.
Card to host, other	No parser cap	The largest reply is STREAM_READ at 6 + 4096 bytes. The card-to-host FIFO is 16 KB.

A host that offers a receive buffer smaller than a reply must read the length bytes first and discard what it cannot keep; the card does not fragment replies.

3.4 Command sequence

1. Check TX_READY (STATUS bit 6) and write the opcode, LEN_LO, LEN_HI, and payload to DATA, one byte per write. The host-to-card FIFO is 4 KB, so a full-length command fits without draining.
2. The card parses the header, waits for LEN payload bytes, and dispatches. A command that completes immediately (PING, HASH, VOL_STAT) has its reply queued before the dispatcher returns. A command that waits on the network (CONNECT, HTTP, VOL_OPEN, VOL_READ on a cache miss, STREAM_OPEN) is accepted and its reply arrives when the work completes.
3. The host reads frames from DATA until it sees the reply opcode. Events may arrive before the reply and must be consumed or discarded (Chapter 5). A reply with a handle or socket byte is matched on that byte as well as the opcode.

The card never interleaves the bytes of two card-to-host frames; each frame is contiguous in the FIFO. The host must finish writing one command before it begins another. Two commands may be outstanding at once when they are independent (an HTTP request on socket 0 and a VOL_READ), but the host must then demultiplex replies by opcode and handle rather than by order.

3.4.1 Worked example: PING

Host writes:

```
00 00 00                      PING, no payload
```

Card replies:

```
80 04 00  01 0D 04 FF         fw_major 1, fw_minor 13, proto $04, caps $FF
```

3.4.2 Worked example: a command with a deferred reply

Host writes VOL_READ for handle 0, block 2:

```
51 05 00  00 02 00 00 00
```

The card answers when the block is in hand, from cache on the next poll or after a network fetch:

```
D1 06 02  00 00 02 00 00 00   handle 0, result 0, block 2
<512 data bytes>
```

The reply length `$0206` is 6 header bytes plus 512 data bytes. On an error the reply is `D1 06 00` with the six header bytes and no data.

3.5 Errors and resynchronization

Two kinds of failure are reported two different ways.

A command that is understood but fails answers with its own reply opcode and a nonzero `result` byte from Appendix A. VOL_READ on an unmounted handle replies `$D1` with result 16, VOL_NOMOUNT. A client waiting on `$D1` therefore always gets one.

A command the parser cannot accept is answered with the ERROR event `$93` carrying sock `$FF` and a code, and no reply of the command's own opcode is sent. The cases are:

- A header whose LEN exceeds 2048: code 10, BAD_FRAME. The parser discards the header and resynchronizes on the next byte as a new opcode.
- A payload of the wrong length for the opcode: code 10, BAD_FRAME.
- An opcode the card does not implement, or a feature the build lacks: code 11, UNSUPPORTED. The disk commands `$50` to `$5B` are the exception; they always answer their own reply opcode with result 11 so a slot ROM waiting on `$Dx` never hangs.
- A socket or service in the wrong state for the command (SEND on a closed socket, NONCE_CANCEL when no scan is running): code 8, SOCKET_STATE.

After any parse error the host should assume the card is at a frame boundary and the host's own view of the stream may not be. The clean recovery is CONTROL bit 3 (flush the host-to-card FIFO and reset the parser), then CONTROL bit 2 (flush the card-to-host FIFO), then read SYNC. SYNC increments when the parser reset completes, so a host that reads it before and after knows the card has re-armed. A soft reset (CONTROL bit 0) does the same and also aborts every command and socket in flight.

3.6 Deferred replies and polling

Commands that touch the network return nothing until the work is done. The host has three ways to wait:

- Poll STATUS bit 7 and read frames as they arrive. This is what the slot ROM and cogslib do.
- Poll RXLO/RXHI for a count large enough to hold the expected reply, then drain with the wide window (Chapter 2).
- For NONCE_SCAN, watch for `$97` NONCE_PROGRESS events (about every two seconds) while waiting on `$B3`.

There is no timeout in the protocol. The card resolves every accepted command with a reply or an ERROR event; a network operation that cannot complete ends with a code such as 3, CONN_TIMEOUT, from the card's own timers. The slot ROM adds its own bounded waits on top so a bus fault cannot hang a boot; a host program may do the same.

3.7 Capability discovery

PING is the feature probe. Its reply carries the firmware major and minor, the protocol version (`$04`, the same value VERSION reads), and a caps byte:

Table 3-3. PING caps byte

Bit	Name	Covers
0	NET	WIFI_*, RESOLVE, CONNECT/SEND/CLOSE
1	TLS	CONNECT flag bit 0, `https://` URLs
2	HTTP	HTTP command
3	HASH	HASH
4	RANDOM	RANDOM
5	JSON	JSON_GET
6	NONCE	NONCE_SCAN, NONCE_CANCEL
7	NETTOOLS	RESOLVE, TCP_PING, ICMP_PING, TRACEROUTE

The card reports `$FF`. Features added after the caps byte filled (the disk commands, streams, CONFIG, the player and picture commands) are probed by call: a card without one answers UNSUPPORTED and the client treats that as absent.

CHAPTER 4

Command Reference

This chapter lists every command the card accepts, grouped by function. Each entry gives the opcode and reply opcode, the request payload, the reply payload, whether the reply is immediate or deferred, and the errors the command returns. Multi-byte integers are little-endian unless marked. String fields are length-prefixed with one byte and are not NUL-terminated. Requests of the wrong length are answered with ERROR code 10, BAD_FRAME, unless the entry says otherwise; that case is not repeated below.

Immediate means the reply is queued before the dispatcher returns. *Deferred* means the card accepts the command and the reply arrives when the operation completes, which may be after network activity.

4.1 Summary

Table 4-1. Command summary

Op	Name	Reply	Timing	Section
`$00`	PING	`$80`	Immediate	System
`$01`	STATUS	`$81`	Immediate	System
`$02`	WIFI_SCAN	`$82`	Deferred	Network
`$03`	WIFI_JOIN	`$83`	Deferred	Network
`$04`	WIFI_SAVE	`$84`	Immediate	Network
`$05`	TIME	`$85`	Immediate	System
`$06`	PREFS_GET	`$86`	Immediate	System
`$07`	PREFS_SET	`$87`	Immediate	System
`$08`	NETINFO	`$88`	Immediate	Network
`$09`	TLS_INFO	`$89`	Immediate	Network
`$0A`	CARD_INFO	`$8A`	Immediate	System
`$0B`	RESOLVE	`$8B`	Deferred	Network tools
`$0C`	TCP_PING	`$8C`	Deferred	Network tools
`$0D`	ICMP_PING	`$8D`	Deferred	Network tools
`$0E`	TRACEROUTE	`$8E`	Deferred	Network tools
`$0F`	LOCALTIME	`$8F`	Immediate	System
`$10`	CONNECT	`$90` event	Deferred	Sockets
`$11`	SEND	`$9A` event	Immediate	Sockets
`$12`	CLOSE	`$92` event	Deferred	Sockets
`$20`	HTTP	`$91`/`$95` events	Deferred	HTTP
`$30`	HASH	`$B0`	Immediate	Card services
`$31`	RANDOM	`$B1`	Immediate	Card services
`$32`	JSON_GET	`$B2`	Immediate	Card services

Op	Name	Reply	Timing	Section
`$33`	NONCE_SCAN	`$B3`	Deferred	Card services
`$34`	NONCE_CANCEL	`$B3`	Immediate	Card services
`$40`	LINK_OPEN	`$C0`	Immediate	Link layer
`$41`	LINK_FRAME	none	Immediate	Link layer
`$42`	LINK_CLOSE	`$C2`	Immediate	Link layer
`$43`	LINK_IP	`$C3`	Immediate	Link layer
`$44`	LINK_POLL	`$C4`	Immediate	Link layer
`$50`	VOL_OPEN	`$D0`	Deferred	Network disk
`$51`	VOL_READ	`$D1`	Deferred	Network disk
`$52`	VOL_CLOSE	`$D2`	Immediate	Network disk
`$53`	VOL_URL_SET	`$D3`	Immediate	Network disk
`$54`	VOL_URL_GET	`$D4`	Immediate	Network disk
`$55`	VOL_MOUNT_SET	`$D5`	Immediate	Network disk
`$56`	VOL_MOUNT_GET	`$D6`	Immediate	Network disk
`$57`	VOL_WRITE	`$D7`	Deferred	Network disk
`$58`	VOL_SWITCHED	`$D8`	Immediate	Network disk
`$59`	CAT_URL_SET	`$D9`	Immediate	Catalog sources
`$5A`	CAT_URL_GET	`$DA`	Immediate	Catalog sources
`$5B`	VOL_STAT	`$DB`	Immediate	Network disk
`$5C`	AUD_URL_SET	`$DC`	Immediate	Catalog sources
`$5D`	AUD_URL_GET	`$DD`	Immediate	Catalog sources
`$5E`	AUD_EQ_SET	`$DE`	Immediate	Player
`$5F`	AUD_EQ_GET	`$DF`	Immediate	Player
`$60`	CONFIG	`$E0`	Immediate	Config terminal
`$61`	CDA_INSTALL	`$E1`	Immediate	Config terminal
`$62`	AUD_EQG_SET	`$E2`	Immediate	Player
`$63`	AUD_EQG_GET	`$E3`	Immediate	Player
`$64`	PIC_URL_SET	`$E4`	Immediate	Catalog sources
`$65`	PIC_URL_GET	`$E5`	Immediate	Catalog sources
`$66`	BOOT_PICK	`$E6`	Immediate	Config terminal
`$67`	CARDCFG_SET	`$E7`	Immediate	System
`$68`	CARDCFG_GET	`$E8`	Immediate	System
`$69`	VOL_EJECT	`$E9`	Immediate	Network disk
`$6A`	SP_SCAN	none	Fire-and-forget	Network disk
`$70`	STREAM_OPEN	`$F0`	Deferred	Byte streams
`$71`	STREAM_READ	`$F1`	Immediate	Byte streams
`$72`	STREAM_CLOSE	`$F2`	Immediate	Byte streams
`$73`	STREAM_STAT	`$F3`	Immediate	Byte streams
`$74`	STREAM_SEEK	`$F4`	Deferred	Byte streams

4.2 System

4.2.1 PING (`$00`, reply `$80`)

Request: none.

Table 4-2. PING reply

Offset	Size	Field
0	1	fw_major
1	1	fw_minor
2	1	proto_ver, `$04`
3	1	caps, the capability bitmap in Chapter 3

4.2.2 STATUS (`$01`, reply `$81`)

Request: none. The reply is 9, 11, or 17 bytes. A client reads the fields it knows and ignores the rest.

Table 4-3. STATUS reply

Offset	Size	Field
0	1	link_state: 0 down, 1 joining, 2 up with address
1	4	ip, a.b.c.d
5	1	rssi, dBm, signed
6	2	free_heap_kb
8	1	active_sockets
9	2	die temperature, tenths of a degree C, signed
11	2	VSYS, millivolts
13	4	uptime, seconds

4.2.3 TIME (`$05`, reply `$85`)

Request: none. Reply: 4 bytes, Unix epoch seconds, UTC. Zero until the card has synchronized with NTP.

4.2.4 LOCALTIME (`$0F`, reply `$8F`)

Request: none.

Table 4-4. LOCALTIME reply

Offset	Size	Field
0	1	valid: 0 no wall clock yet, do not write
1	1	auto: the auto-sync flag from the prefs blob
2	4	secs: local time as seconds since 1904-01-01

`secs` is UTC plus the timezone offset stored in the prefs blob plus 2 082 844 800, the 1904-to-1970 epoch difference, ready to store into the IIGS clock chip.

4.2.5 PREFS_GET (`$06`, reply `$86`) and PREFS_SET (`$07`, reply `$87`)

A 32-byte settings area the card persists in flash on behalf of host software.

PREFS_GET request: none. Reply: result, then the stored bytes (zero bytes when nothing is stored).

PREFS_SET request: the whole blob, at most 32 bytes. Reply: result.

The card interprets the first 15 bytes; host software may use bytes 15 to 31 freely.

Table 4-5. Prefs blob layout

Byte	Field
0	Layout version, 4
1	Auto-sync clock on join, 0 or 1
2 to 3	Timezone offset, minutes, signed
4	TLS flags: bit 0 skip certificate verification
5	Bass shelf gain, dB plus 12
6	Treble shelf gain, dB plus 12
7 to 14	Graphic EQ band gains, dB plus 12, 60 Hz to 10 kHz

Gain bytes are stored biased by 12 so that 12 is flat and 0 is a 12 dB cut. A host that grows an older blob must initialize new gain bytes to 12, not 0.

4.2.6 CARD_INFO (`$0A`, reply `$8A`)

Request: none.

Table 4-6. CARD_INFO reply

Offset	Size	Field
0	8	Board serial number
8	1	Last reset reason
9	4	System clock, Hz

4.2.7 CARDCFG_SET (`$67`, reply `$E7`) and CARDCFG_GET (`$68`, reply `$E8`)

The card's own network address and boot chooser setting, a 22-byte record separate from the prefs blob.

CARDCFG_SET request: the 22-byte record. Reply: result.

CARDCFG_GET request: none. Reply: result, then the record.

Table 4-7. Card settings record

Offset	Size	Field
0	1	ip_mode: 0 DHCP, nonzero static
1	4	ip
5	4	mask
9	4	gateway
13	4	dns1, all zero means the gateway
17	4	dns2, all zero means none
21	1	boot_menu: 0 boot at once, else the chooser countdown in seconds

A static address applies to the medium carrying traffic (wired when the cable is up, Wi-Fi otherwise). The other medium keeps a DHCP lease.

4.3 Network

4.3.1 WIFI_SCAN (`$02`, reply `$82`)

Request: none. The card scans for a few seconds and replies once. If a scan is already running the reply is ERROR code 15, BUSY; during a join it is code 8, SOCKET_STATE.

Table 4-8. WIFI_SCAN reply, one record per network

Offset	Size	Field
0	1	count
+0	1	rssi, dBm, signed
+1	1	auth: 0 open, 1 secured
+2	1	ssid_len, 1 to 32
+3	ssid_len	ssid

Networks are strongest first and deduplicated by name.

4.3.2 WIFI_JOIN (`$03`, reply `$83`) and WIFI_SAVE (`$04`, reply `$84`)

Request for both:

Table 4-9. WIFI_JOIN and WIFI_SAVE request

Offset	Size	Field
0	1	ssid_len, 1 to 32
1	ssid_len	ssid
+0	1	psk_len, 0 to 63
+1	psk_len	psk

Reply: result. WIFI_JOIN associates with the network; the security mode is chosen from what the last scan reported for that SSID, WPA2-AES when the name was not in the scan and a key is given, open otherwise. WIFI_SAVE writes the credentials to flash so the card joins at power-on; it does not join. Link state changes are reported by the WIFI_STATE event.

4.3.3 NETINFO (`$08`, reply `$88`)

Request: none. The first nine bytes mirror STATUS.

Table 4-10. NETINFO reply

Offset	Size	Field
0	1	link_state
1	4	ip
5	4	subnet
9	4	gateway
13	4	dns1
17	4	dns2, 0.0.0.0 if unset
21	6	mac, most significant byte first. The MAC the link layer must transmit with: the STA MAC when Wi-Fi routes; when the wire routes (firmware 1.26) `06:A2:` followed by the card's wired MAC tail, the GS's own address on the LAN, so the router hands out a second lease and the two hosts have two MACs. Card Status keeps showing the card's own MAC.
27	1	rssi, dBm, signed
28	1	ssid_len, 0 when wired or not associated
29	ssid_len	ssid
+0	1	ll_active: 1 while LINK_OPEN is held
+1	1	ll_mode: 0 bridge, 1 NAT; bit 7 set while the link is in poll mode
+2	4	gs_ip: the address reported by LINK_IP, 0.0.0.0 if none

The identity reported is that of the medium currently routing traffic. `ssid_len` 0 with `link_state` 2 means the wired port.

4.3.4 TLS_INFO (`$09`, reply `$89`)

Request: none. Describes the most recent TLS session the card completed.

Table 4-11. TLS_INFO reply

Offset	Size	Field
0	1	valid: 0 if no TLS session since power-on
1	1	verified: 1 if the certificate chain validated
2	1 + n	version, length-prefixed, "TLSv1.2"
	1 + n	cipher suite name, length-prefixed
	1 + n	peer certificate subject CN, length-prefixed

4.4 Network tools

All four are covered by caps bit 7. Each is a single request and a single reply; the card holds no state between calls.

4.4.1 RESOLVE (`$0B`, reply `$8B`)

Request: the hostname, 1 to 255 bytes, no length prefix (the whole payload).

Table 4-12. RESOLVE reply

Offset	Size	Field
0	1	result: 0, or 1 NOT_JOINED, 2 DNS_FAIL
1	1	naddr, 0 to 4
2	4 each	IPv4 addresses

4.4.2 TCP_PING (`$0C`, reply `$8C`)

Request: port (2 bytes), then the hostname (the rest of the payload). The card resolves, times a TCP connect, and closes the socket.

Table 4-13. TCP_PING reply

Offset	Size	Field
0	1	result: 0 connected, 2 DNS_FAIL, 3 CONN_TIMEOUT, 4 CONN_REFUSED, 1 NOT_JOINED
1	2	rtt_ms, valid when result is 0
3	4	Resolved address, 0.0.0.0 if DNS failed

Result 4 means the host answered with a reset: reachable, nothing listening on that port.

4.4.3 ICMP_PING (`$0D`, reply `$8D`)

Request: the hostname, 1 to 255 bytes. Reply: identical in shape to TCP_PING. Result 3 means no echo reply within the window.

4.4.4 TRACEROUTE (`$0E`, reply `$8E`)

Request: ttl (1 byte, 1 to 64), then the hostname (1 to 254 bytes). Each call probes exactly one hop; the client loops ttl upward and stops on the DONE flag or its own hop cap.

Table 4-14. TRACEROUTE reply

Offset	Size	Field
0	1	result: 0 probe ran, else 1 NOT_JOINED, 2 DNS_FAIL, 12 INTERNAL
1	1	ttl, echoed
2	4	Responding hop, 0.0.0.0 on timeout
6	2	rtt_ms, valid unless TIMEOUT
8	1	flags: bit 0 DONE (this hop is the destination), bit 1 TIMEOUT

4.5 Sockets

Four sockets, numbered 0 to 3. At most two may be TLS at once; a third TLS CONNECT fails with 7, NO_FREE_SOCK. Plain TCP may use all four. Socket 3 is used by the card's own Firmware Update page while that page is open.

4.5.1 CONNECT (`$10`, completes with CONNECTED `$90`)

Table 4-15. CONNECT request

Offset	Size	Field
0	1	sock, 0 to 3
1	1	flags: bit 0 TLS, bit 1 skip certificate verification
2	2	port
4	1	host_len
5	host_len	host

The reply is the CONNECTED event: sock, result. On a nonzero result the socket is closed. A verified TLS connect (bit 0 set, bit 1 clear) requires a synchronized clock; while TIME reads zero the connect is refused at once with 5, TLS_HANDSHAKE. SNI is always sent.

4.5.2 SEND (`$11`, acknowledged by SEND_ACK `$9A`)

Request: sock, then the data (the rest of the payload, at most 2047 bytes). The acknowledgement is SEND_ACK: sock, result. Result 0 means the bytes were queued. Sending on a socket that is not connected returns 8, SOCKET_STATE.

Received bytes arrive as DATA events (Chapter 5). The card buffers up to 16 KB per socket and applies TCP flow control to the remote end when the host drains slowly.

4.5.3 CLOSE (`$12`, completes with CLOSED `$92`)

Request: sock. The reply is the CLOSED event: sock, reason.

4.6 HTTP

4.6.1 HTTP (`$20`, completes with DATA `$91` events and HTTP_DONE `$95`)

One command performs DNS, TCP, TLS when the URL is `https://`, the request, and the response.

Table 4-16. HTTP request

Offset	Size	Field
0	1	sock, 0 to 3
1	1	flags, see the HTTP flags table
2	1	method_len
3	method_len	method, “GET”, “POST”, …
+0	1	url_len
+1	url_len	url, absolute
+0	2	hdrs_len
+2	hdrs_len	Extra header lines, CRLF or LF separated
+0	2	body_len
+2	body_len	Request body

Table 4-17. HTTP flags

Bit	Name	Meaning
0	STREAM	Deliver the body as it arrives instead of after completion.
1	HDRECHO	Include the response header lines in the first DATA event. Ignored when RETAIN is set.
2	RETAIN	Keep the decoded body on the card for JSON_GET instead of delivering it.
3	KEEPALIVE	Park the connection for the next request to the same host.
4	DISCARD	Count the body and deliver nothing.

The card adds `Host`, `Connection: close` (or `keep-alive`), `Accept-Encoding: identity`, `User-Agent`, and `Content-Length` when a body is present. The User-Agent is `CoGS/<version> <board> id/<card ID>`, where the card ID is the first 32 bits of SHA-256 over `CoGS` followed by the chip's 64-bit unique id, as eight hex digits; the same value is on the About page and the Support Snapshot. It removes chunked transfer encoding, so DATA events carry decoded body bytes only. A response with neither `Content-Length` nor chunking ends at the remote close. Up to three redirects are followed.

Completion is exactly one of:

- Zero or more DATA events, then one HTTP_DONE with sock, http_status, total_len. `total_len` equals the sum of the DATA payload lengths.
- One ERROR event with the socket number and a code, and no HTTP_DONE.

A retained body is limited to 32 KB. A larger body ends with ERROR code 9, BUF_OVERRUN, and nothing is retained.

4.7 Card services

The four services run on the card with no network. HASH, RANDOM, and NONCE_SCAN work with the link down; JSON_GET needs a retained HTTP response.

4.7.1 HASH (`$30`, reply `$B0`)

Table 4-18. HASH request

Offset	Size	Field
0	1	algo: 0 SHA-256, 1 double SHA-256
1	2	data_len, must equal the remaining payload
3	data_len	data

Reply: the 32-byte digest, most significant byte first (the order `hashlib.sha256().digest()` produces). Zero-length data is valid. The hash runs on the RP2350 SHA-256 peripheral.

4.7.2 RANDOM (`$31`, reply `$B1`)

Request: count, 1 to 255. Reply: count bytes from the hardware true random number generator. Count 0 is BAD_FRAME.

4.7.3 JSON_GET (`$32`, reply `$B2`)

Reads one value out of the document retained by the last HTTP command that set RETAIN on the same socket.

Table 4-19. JSON_GET request

Offset	Size	Field
0	1	sock
1	1	path_len
2	path_len	path

The path is a dot and bracket expression: `a.b.c`, `arr[0]`, `choices[0].message.content`. An empty path returns the whole root value. Keys are matched as raw bytes.

Table 4-20. JSON_GET reply, repeated per chunk

Offset	Size	Field
0	1	result: 0, 13 NO_RETAINED, 14 JSON_NOPATH
1	2	value_len, the total decoded length
3	up to 1024	value chunk

The value is delivered in chunks of at most 1024 bytes; `value_len` is repeated in every chunk and the client concatenates until it holds that many bytes. A found value always produces at least one frame. Strings are returned decoded to UTF-8 with escapes resolved; numbers, booleans, and null as their literal token; objects and arrays as their raw JSON text. A chunk boundary never splits a multi-byte UTF-8 sequence. The retained document persists until the next retaining HTTP command replaces it.

4.7.4 NONCE_SCAN (`$33`, deferred reply `$B3`) and NONCE_CANCEL (`$34`)

Table 4-21. NONCE_SCAN request, 116 bytes

Offset	Size	Field
0	76	header, a block header without its nonce
76	32	target, most significant byte first
108	4	nonce_start
112	4	nonce_count, nonzero

For each nonce from nonce_start, the card computes double SHA-256 of the header with the nonce appended little-endian and compares the digest to the target as 256-bit big-endian integers. The scan runs cooperatively between network polls and emits NONCE_PROGRESS about every two seconds. A second NONCE_SCAN while one runs returns ERROR 15, BUSY.

Table 4-22. NONCE_SCAN reply

Offset	Size	Field
0	1	result: 0 found, 1 exhausted, 2 cancelled
1	4	winning_nonce
5	4	hashes_done

NONCE_CANCEL takes no payload and aborts the running scan. The scan's own `$B3` with result 2 is the response. NONCE_CANCEL with no scan running returns ERROR 8, SOCKET_STATE.

4.8 Link layer

The link layer carries a host TCP/IP stack's own frames through the card so the Apple II is a host on the LAN in its own right. It is used by the Marinetti link layer on the IIGS and the IP65 driver on 8-bit machines.

4.8.1 LINK_OPEN (`$40`, reply `$C0`)

Request: mode, 0 bridge or 1 NAT, optionally OR `$80` for poll mode (firmware 1.26, link layer 1.3). Reply: result. In bridge mode the card forwards raw Ethernet frames and the host stack runs DHCP against the LAN router. In NAT mode the host uses 192.168.42.2/24 with gateway and DNS 192.168.42.1 and the card translates; the card forwards exactly the IP datagram (an Ethernet pad from the wire is stripped) and recomputes the TCP or UDP checksum in place after rewriting the address.

The bridge sits on the link that routes at OPEN time: the W5500 when the cable is up with an address, else the Wi-Fi STA. On the wire the GS transmits with its own MAC (see NETINFO) and the W5500's hardware MAC filter is turned off for the session; inbound unicast is demultiplexed by destination MAC (the GS's goes to the host and never to lwIP, the card's is never mirrored). On Wi-Fi the two hosts share the STA MAC and are told apart by IP. Broadcast and ARP go to both. IPv4 multicast (`01:00:5E:...`) is not mirrored; Marinetti and IP65 have no multicast receive, and LAN mDNS/SSDP chatter was crossing the bus at 17 frames a second. Unicast to the card's own IP is never mirrored.

Poll mode. Without it inbound frames arrive as `$96` events on the same card-to-host ring as the block driver's `$D1` and `$D7` replies, and every slot ROM reply loop (WAIT_RESP, the STATUS paths) pops and discards whatever is not the reply it wants, so on a live desktop (Finder unit polls, block reads) queued link frames were thrown away. With bit 7 set the card queues inbound frames in one 17,920-byte pool slot (`[len lo][len hi][frame]` byte ring; the newest frame is dropped and counted when it is full) and emits no `$96`. The host fetches one per LINK_POLL from its own poll entry, gated by a read of LINKQ at register +4, so frames are only ever dequeued inside the link layer's own request and reply exchange and the slot ROM never sees them. The slot is taken at OPEN (an idle volume read-ahead window is evicted if none is free) and returned at CLOSE, which logs the queue's high-water mark and drops. Firmware before 1.26 rejects the flag with E_BAD_FRAME before any teardown and link layer 1.3 reopens plain; 1.26 with no free slot returns E_BUSY, same retry. Only the link layer is in this path; block traffic, streams, mounts, TLS, OTA, the CDA and the NDA/CDev are untouched.

4.8.2 LINK_FRAME (`$41`, no reply)

Request: one outbound frame (bridge) or IP packet (NAT). Inbound traffic arrives as LINK_FRAME events `$96`. Frames beyond the card's buffer are dropped, as on any network interface.

4.8.3 LINK_CLOSE (`$42`, reply `$C2`)

Request: none. Reply: result.

4.8.4 LINK_IP (`$43`, reply `$C3`)

Request: the host's IPv4 address, 4 bytes, reported after DHCP or static configuration so NETINFO can show it. Reply: result.

4.8.5 LINK_POLL (`$44`, reply `$C4`)

Request: none. Reply: the oldest queued inbound frame (bridge) or IP packet (NAT) as the whole payload, or an empty payload when the queue is empty. Only meaningful while a poll-mode LINK_OPEN is held; otherwise the reply is empty. Firmware 1.26.

4.9 Network disk

The card mounts a disk image at an HTTP or HTTPS URL as a block device. Handles are 0 to 7 and correspond to SmartPort units 1 to 8 (unit = handle + 1). Blocks are 512 bytes and numbered from 0; a

2IMG header, when present, is skipped so block 0 is the first ProDOS block. Chapter 6 describes the fetch and cache path.

4.9.1 VOL_OPEN (`$50`, reply `$D0`)

Table 4-23. VOL_OPEN request

Offset	Size	Field
0	1	handle, 0 to 7
1	1	flags, reserved, 0
2	1	url_len
3	url_len	url

Reply: handle, result, total_blocks (4 bytes). The card resolves, connects, reads the image header, and replies. Redirects are followed.

4.9.2 VOL_READ (`$51`, reply `$D1`)

Request: handle, block (4 bytes), and an optional flags byte. Flags bit 0, VOLRD_WIN, asks for delivery through the `$C800` block window (Chapter 6).

Reply: handle, result, block (4 bytes), then 512 data bytes. On error, or when the block went to the window, the reply is the 6-byte header alone. A malformed request (length not 5 or 6, or a bad handle) is answered with a data-less `$D1` carrying BAD_FRAME, not an ERROR event.

A read on a handle with no open volume mounts the URL stored for that unit first (VOL_URL_SET or VOL_MOUNT_SET); result 16, VOL_NOMOUNT, means no URL is stored. Result 17, VOL_RANGE, means the block is past the end of the volume.

4.9.3 VOL_WRITE (`$57`, reply `$D7`)

Two request forms:

- handle, block (4 bytes), 512 data bytes: the data is in the frame.
- handle, block (4 bytes), flags: with flags bit 0, VOLWR_WIN, the 512 bytes are taken from the `$C800` block window, which the host filled before sending this header.

Reply: handle, result, block (4 bytes). The card writes the block to the origin with a ranged HTTP PUT and invalidates cached copies. Result 19, WIN_SHORT, means fewer than 512 bytes were stored into the window and nothing was written; the host resends with the data in the frame. A write the origin rejects (HTTP 403, 405, 501) latches the mount read-only; VOL_STAT then reports the flag and later writes fail immediately with 11, UNSUPPORTED, without network traffic.

4.9.4 VOL_CLOSE (`$52`, reply `$D2`)

Request: handle. Reply: handle, result. Closes the connection and frees the volume's read-ahead window.

4.9.5 VOL_STAT (`$5B`, reply `$DB`)

Request: handle. Reply: handle, result, total_blocks (4 bytes), flags. Flags bit 0, VOLF_RO, means the volume is read-only. Answers from the mount table without block I/O; mounts from the stored URL on first touch like VOL_READ. A unit under the two-phase remount hold (Chapter 6) answers 16, VOL_NOMOUNT, once and releases the hold; VOL_READ and VOL_WRITE do the same.

4.9.6 VOL_URL_SET (`$53`, reply `$D3`) and VOL_URL_GET (`$54`, reply `$D4`)

The stored URL for unit 1, the boot volume. VOL_URL_SET request: url_len, url (url_len 0 clears). Reply: result. VOL_URL_GET request: none. Reply: url_len, url. The maximum URL length is 255. A successful set also records the URL on the recent-disk list (Chapter 6).

4.9.7 VOL_MOUNT_SET (`$55`, reply `$D5`) and VOL_MOUNT_GET (`$56`, reply `$D6`)

The stored URL for any unit. VOL_MOUNT_SET request: unit (1 to 8), url_len, url (url_len 0 clears the unit). Reply: result. Setting a unit that is mounted applies at once: the card drops the open volume, the next read mounts the new URL, and the unit's disk-switched flag is raised. While a GS/OS desktop is live the dropped unit is also held empty until GS/OS's next STATUS or read has seen it so (Chapter 6, two-phase remount). Result 20, VOL_DUP, means that image is already assigned to another unit. A successful set records the URL on the recent-disk list the same way VOL_URL_SET does.

VOL_MOUNT_GET request: unit. Reply: result, unit, url_len, url.

4.9.8 VOL_SWITCHED (`$58`, reply `$D8`)

Request: unit. Reply: result, unit, switched. Reads and clears the unit's disk-switched flag. The slot ROM reports the flag in the SmartPort status byte so the operating system remounts the volume.

4.9.9 VOL_EJECT (`$69`, reply `$E9`)

Request: unit (1 to 8). Reply: result, unit. Ejects the image from the unit. Firmware 1.23; the slot ROM sends it for SmartPort CONTROL code 4 (Eject), which is what GS/OS issues from the Finder's Eject, from the "not a recognized disk" dialog, and from Shut Down, which ejects every device whose DIB says removable media. Other CONTROL codes are answered no-op OK by the ROM without a frame. A unit outside 1 to 8 answers 10, BAD_FRAME. VOL_EJECT clears any two-phase remount hold on the unit: GS/OS asked for the eject and knows the unit is empty. The DIB device type is $06 from 1.23 on (was $02, hard disk): GS/OS's generated SmartPort driver forwards Eject only to DIB types $01, $05, $06 and $07, and $06 is the one that costs no alert dialog, no cache change and no icon change.

From 1.29 the eject is pending rather than final. The mount is dropped at once (window freed, sockets closed, read-only latch cleared), the unit answers offline with 0 blocks to STATUS and no volume to READ, and the stored URL is left alone. What settles it is the next host frame that is not itself a VOL_EJECT. Any ordinary call (a STATUS poll, a block read, a CDA frame) means the session went on, so the eject was a user's: the stored URL is cleared then, the 1.23 to 1.28 result. A boot marker instead lifts every pending eject with the mounts kept: SP_SCAN (below), BOOT_PICK, the boot-time CONFIG, or a bus RESET. That is what tells a Shut Down from a Finder Eject without a timer: GS/OS polls a unit it has just ejected straight away, and after Shut Down's ejects nothing reaches the card until the next boot. Firmware 1.23 to 1.28 cleared the stored URL on every eject, so a Shut Down emptied all eight units. Chapter 6, "Pending eject".

4.9.10 SP_SCAN (`$6A`, no reply)

Request: one byte, the shape of the SmartPort STATUS the slot ROM is serving. No reply on any path (fire-and-forget, like LINK_FRAME); the ROM sends it from inside the STATUS and has nothing to drain.

Payload 0 (firmware 1.30) is STATUS code 0 for unit 0, the device-count call, sent right after the count is stored. GS/OS's driver generation makes that call once per boot before it DIBs any unit, ProDOS 8′s slot scan makes it too, and nothing on a running desktop makes it again, so the card takes it as a boot: every pending VOL_EJECT is lifted, mounts kept. A boot with nothing pending, or an 8-bit host, is a no-op. It exists because GS/OS's Restart (the button on the "You may now switch off" screen, or a key there) is a software reset, `sta $C068` then `jmp $FA62`, with no RESET pulse on the bus; a card in a slot other than the boot slot otherwise saw nothing of the new boot before GS/OS's first per-unit STATUS, which it could not tell from the Finder's poll after a real eject.

Payload 1 + status code (firmware 1.31) is a per-unit STATUS: 1 for code 0 (device status), 4 for code 3 (the DIB). These never touch an eject. They tell the card which boot the unit-0 call belongs to. GS/OS's driver generation follows the count with a DIB per unit (`build_class1`); ProDOS 8′s loader follows it with a code-0 STATUS per unit (`new_mount`), and it runs again whenever a P8 application is launched from the Finder, with the card's desk accessory still resident. The card arms on payload 0 and lets the first per-

unit marker decide: a DIB means a GS/OS boot with a fresh Desk Manager, so the CDA register (`+8`) is cleared along with the desktop-seen flag and the CDA reply counter, the same three a bus RESET clears, and the desktop-live flag is dropped as a boot-slot BOOT_PICK drops it; a code `0` means ProDOS 8, and the latch is left alone, so a P8 excursion does not put a second CoGS Config in the menu. Anything but the ROM's own VOL_STAT and VOL_SWITCHED between the two markers disarms. A DIB on its own means nothing: the generated driver's status path is a DIB, and the Finder makes hundreds of them on a live desktop. Before 1.31 the latch survived GS/OS's Restart (a software reset) and `CDA_CALL` refused to install into the new Desk Manager until a power cycle.

4.10 Catalog sources

Six commands store the base URLs the card's Browse pages read their catalogs from. Each SET takes url_len, url and replies with result; url_len 0 reverts to the a2cogs default. Each GET takes no payload and replies url_len, url.

Table 4-24. Catalog base URLs

Catalog	Set	Get	Default
Disk images	CAT_URL_SET `$59`	CAT_URL_GET `$5A`	`https://a2cogs.com/disks/`
Audio	AUD_URL_SET `$5C`	AUD_URL_GET `$5D`	`https://a2cogs.com/audio/`
Pictures	PIC_URL_SET `$64`	PIC_URL_GET `$65`	`https://a2cogs.com/pictures/`

Appendix B gives the catalog file formats.

4.11 Byte streams

A pull-model byte stream. The card fetches from the source into a 32 KB ring and stops reading the source when the ring is full, so TCP flow control paces the origin to the host's rate. The host reads at its own pace with no timing obligation. Handle 0 is the network stream; handle 1 is the card-internal `tool://` stream (Chapter 7). One network stream is open at a time; a second STREAM_OPEN returns 15, BUSY.

4.11.1 STREAM_OPEN (`$70`, reply `$F0`)

Table 4-25. STREAM_OPEN request

Offset	Size	Field
0	1	url_len
1	url_len	url
+0	4	offset, optional: open an HTTP source at this byte with a Range request
+4	1	wire, optional, see the wire byte table

Sources are `http://` and `https://` (the GET body), `tcp://host:port` (the raw socket), and `tool://`. Reply: handle, result. For HTTP the reply arrives after the response header parses; a non-2xx status is 4, CONN_REFUSED, and a nonzero offset the server does not honor with 206 is an error.

An HTTP response with `Content-Type audio/mpeg` (also `audio/mp3`, `audio/x-mpeg`), or an `ICY 200 OK` status line with no content type, is decoded on the card and delivered as unsigned 8-bit mono PCM at 21 973 Hz with `$00` samples raised to `$01`. Other declared audio types (AAC, Ogg, FLAC, Opus) fail the open with 11, UNSUPPORTED. Any other content passes through unchanged.

Table 4-26. STREAM_OPEN wire byte

Bit	Name	Meaning
0	STEREO	The file is planar stereo: 16 KB blocks of 8 KB left then 8 KB right. The EQ runs separate state per plane.
1	HIRES	The stereo block shape carries fine and coarse planes of one channel; the EQ is bypassed. Set with STEREO.
2	NOEQ	Raw non-audio bytes (pictures). The EQ passes them untouched and `$00` is not raised.
3	DOWNMIX	The file is planar stereo but the ring emits mono; each left/right pair is averaged. Never set with STEREO.

4.11.2 STREAM_READ (`$71`, reply `$F1`)

Request: handle, max_len (2 bytes). Never blocks.

Table 4-27. STREAM_READ reply

Offset	Size	Field
0	1	handle
1	1	result
2	1	flags: bit 0 FLOWING, bit 1 END, bit 2 STALLED
3	1	level: ring fill, 0 to 255
4	2	len, 0 to min(max_len, 4096)
6	len	data

FLOWING means the source is still open. END means the source has ended and the ring is empty. STALLED means the ring is empty while the source is open, that is, the origin is not keeping up. `len` may be 0 while the ring refills.

4.11.3 STREAM_SEEK (`$74`, reply `$F4`)

Request: handle (0 only), offset (4 bytes). The card reopens the remembered URL with a Range request at the offset, keeping the wire layout. Reply: handle, result, when the reopened source is flowing. `tcp://` and `tool://` sources return 11, UNSUPPORTED. On a DOWNMIX stream the offset is in output (mono) bytes; the card doubles it and rounds to a plane boundary.

4.11.4 STREAM_CLOSE (`$72`, reply `$F2`)

Request: handle. Reply: handle, result. Closes the source and frees the ring.

4.11.5 STREAM_STAT (`$73`, reply `$F3`)

Request: handle.

Table 4-28. STREAM_STAT reply

Offset	Size	Field
0	1	handle
1	1	result
2	1	state: 0 closed, 1 connecting, 2 flowing, 3 ended, 4 error
3	1	level
4	4	buffered, bytes in the ring
8	4	total, bytes delivered since open

Result 18, STREAM_NONE, on READ, SEEK, CLOSE, or STAT means no stream is open on that handle.

4.12 Player

The equalizer runs on the card as the bytes leave the ring for the host, so a change takes effect after the host's own buffered lead. Settings persist in the prefs blob and apply at power-on. Flat settings bypass the filters.

4.12.1 AUD_EQ_SET (`$5E`, reply `$DE`) and AUD_EQ_GET (`$5F`, reply `$DF`)

Two shelving filters: low shelf at 200 Hz, high shelf at 3 kHz. SET request: bass, treble, each a signed dB value clamped to plus or minus 12 in 2 dB steps. Reply: result. GET request: none. Reply: result, bass, treble.

4.12.2 AUD_EQG_SET (`$62`, reply `$E2`) and AUD_EQG_GET (`$63`, reply `$E3`)

Eight peaking filters at 60, 150, 400, 800, 1500, 3000, 6000, and 10 000 Hz, Q 1.4, applied after the shelves. SET request: eight signed dB values, same clamp. Reply: result. GET request: none. Reply: result, eight values.

4.13 Config terminal

The card renders its own menus. The host is a 24 by 40 text screen and a keyboard; all menu logic runs on the card.

4.13.1 CONFIG (`$60`, reply `$E0`)

Table 4-29. CONFIG request

Offset	Size	Field
0	1	key: an Apple keyboard byte with bit 7 set, 0 for the first paint, or a request key code from the table below
1	1	units, optional: the unit count the host presents, so drive pages clamp
2	1	charset, optional: nonzero when the host has lowercase
3	1	slot, optional: the card's slot, 1 to 7, for drive labels
4	1	clock, optional: 0 no writable clock, 1 IIGS clock chip
5	1	source, optional: 0 slot ROM, 1 the card-resident desk accessory

Table 4-30. CONFIG request key codes

Code	Name	Meaning
`$01`	TICK	No key; repaint. Sent while the card runs an asynchronous operation.
`$02`	CLKOK	The SETCLK write succeeded.
`$03`	CLKERR	The SETCLK write failed.
`$04`	PLAYOK	The local player finished (track end).
`$05`	PLAYERR	The local player could not start.
`$06`	PLAYBG	The player moved to the background; answer QUIT.
`$07`	PLAYNEXT	The user skipped forward; advance the queue.
`$08`	PLAYSTOP	The user stopped; do not advance the queue.
`$09`	PLAYPREV	The user stepped back one track.
`$0A`	PLAYMEM	PLAYERR whose cause was an allocation failure.

Reply: flags, then 960 bytes of Apple II text screen codes (24 rows of 40, primary character set, inverse on the selected row).

Table 4-31. CONFIG reply flags

Bit	Name	Meaning
0	QUIT	Leave the menu.
1	POLL	Re-invoke with TICK without waiting for a key.
2	BOOT	With QUIT: the quit was "boot now". With POLL and without QUIT: poll about once a second.
3	SETCLK	Fetch LOCALTIME, write the system clock, report with CLKOK or CLKERR.
4	HALT	A firmware update is committed and the card is about to restart. Stop all card I/O and stop executing card-served code.
5	PLAY	The card opened a byte stream on the picked audio track; this screen is the player page. Run a local player and report with a PLAY code. Raised only for source 1.
6	STEREO	With PLAY: the stream is planar stereo.
7	LIVESRC	With PLAY: the source is live (radio), not seekable.

4.13.2 CDA_INSTALL (`$61`, reply `$E1`)

Request: none or one selector byte. Selector 1 returns the position-independent stage-2 image of the card-resident desk accessory (15 135 bytes); no payload or selector 0 returns the stage-1 installer (831 bytes). The reply is preceded by a 3-byte preamble `C5 A5 3C` ahead of the normal header so a reader can find the frame start by scanning. Since 1.24 the frame is staged in a card buffer slot shared with the volume windows; if every slot is mid-fetch the card answers `E_BUSY` and the requester retries. Chapter 7 describes the installer.

4.13.3 BOOT_PICK (`$66`, reply `$E6`)

Request: unit. 0 or 1 selects the stored order; 2 to 8 serves that unit as drive 1 for this session (a swap of the mount table, not persisted). The slot ROM sends it once per boot with the digit held at power-on. Reply: flags; bit 0 asks the ROM to enter the config terminal, which opens on the boot chooser page.

CHAPTER 5

Event Reference

Events are frames the card sends without a matching request in the host's view: connection state, incoming socket data, errors, and completion notices. They share the card-to-host FIFO with replies and are never interleaved with them. A host that is waiting for a specific reply must be prepared to receive an event first and either act on it or discard it.

5.1 Summary

Table 5-1. Events

Op	Name	Payload
`$90`	CONNECTED	sock, result
`$91`	DATA	sock, bytes
`$92`	CLOSED	sock, reason
`$93`	ERROR	sock, code, msg_len, msg
`$94`	WIFI_STATE	link_state
`$95`	HTTP_DONE	sock, http_status (2), total_len (4)
`$96`	LINK_FRAME	one inbound frame or packet
`$97`	NONCE_PROGRESS	hashes_done (4), rate_hps (4)
`$9A`	SEND_ACK	sock, result

5.2 CONNECTED (`$90`)

Payload: sock, result. Completes a CONNECT. Result 0 means the socket is open and may be used with SEND; the card will deliver received bytes as DATA events. A nonzero result is a code from Appendix A and the socket is closed. Common results are 2 DNS_FAIL, 3 CONN_TIMEOUT, 4 CONN_REFUSED, 5 TLS_HANDSHAKE, and 6 CERT_VERIFY.

5.3 DATA (`$91`)

Payload: sock, then 1 to 1024 bytes. Received bytes from a TCP socket opened with CONNECT, or body bytes from an HTTP command on that socket. The 1024-byte cap is fixed so a host may use a small buffer. The card emits as many DATA events as it takes.

For an HTTP command, DATA carries decoded body bytes: chunked transfer encoding is removed and, unless HDRECHO was set, the response headers are not included. With STREAM set the events arrive as the body does; without it they arrive after the response is complete. With RETAIN set no DATA events are emitted for that request.

5.4 CLOSED (`$92`)

Payload: sock, reason. The socket is closed. Emitted in reply to CLOSE and also when the remote end closes the connection. Reason 0 is an orderly close; other values are codes from Appendix A.

5.5 ERROR (`$93`)

Table 5-2. ERROR payload

Offset	Size	Field
0	1	sock, or `$FF` when the error is not tied to a socket
1	1	code, Appendix A
2	1	msg_len
3	msg_len	ASCII message, diagnostic text

ERROR is emitted in two situations. With sock `$FF` it reports a command the parser or dispatcher could not accept: BAD_FRAME, UNSUPPORTED, SOCKET_STATE, BUSY (Chapter 3). With a socket number it terminates an HTTP command on that socket in place of HTTP_DONE. The message text is for logs and screens; a program keys off the code.

Emitting an ERROR sets STATUS bit 3. The bit stays set until the host reads the pending ERROR event from the FIFO.

5.6 WIFI_STATE (`$94`)

Payload: link_state, with the same values as STATUS: 0 down, 1 joining, 2 up with an address. Emitted when the state changes, including after WIFI_JOIN and when the card joins from stored credentials at power-on. A client that shows link status may poll STATUS instead of consuming this event.

5.7 HTTP_DONE (`$95`)

Table 5-3. HTTP_DONE payload

Offset	Size	Field
0	1	sock
1	2	HTTP status code, 200, 404, …
3	4	total_len, decoded body bytes delivered or retained

Exactly one HTTP_DONE ends every HTTP command that reached a complete response, whatever the status code. `total_len` equals the sum of the DATA payload lengths for that request, or the retained length when RETAIN was set. A command that fails before the response completes ends with ERROR and no HTTP_DONE. A client waits for either HTTP_DONE or ERROR carrying its socket number and needs no timeout of its own.

5.8 LINK_FRAME (`$96`)

Payload: one inbound Ethernet frame (bridge mode) or IP packet (NAT mode), delivered while a plain (non-poll) LINK_OPEN is held. The host stack consumes it as it would a frame from a network interface. Frames that arrive when the card's buffer is full are dropped. A link opened in poll mode (LINK_OPEN bit 7, firmware 1.26) emits no `$96`; the host fetches frames with LINK_POLL instead, because this event rides the same ring as the block driver's replies and the slot ROM's reply loops discard it (see Demultiplexing below).

5.9 NONCE_PROGRESS (`$97`)

Payload: hashes_done (4 bytes), rate_hps (4 bytes, hashes per second). Emitted about every two seconds while a NONCE_SCAN runs, as a liveness signal and for a progress display. The scan ends with its `$B3` reply, not with an event.

5.10 SEND_ACK (`$9A`)

Payload: sock, result. Acknowledges a SEND. Result `0` means the bytes were queued for transmission. It is an event opcode rather than `$91` because `$91` is DATA: a two-byte DATA payload would otherwise be indistinguishable from an acknowledgement.

5.11 Demultiplexing

A client that has more than one thing in flight sorts incoming frames by opcode and then by the first payload byte, which is the socket or handle for every command that has one. The pattern is:

1. Read the opcode and length.
2. If the opcode is the reply being waited for and the socket or handle matches, consume it.
3. If the opcode is an event the client handles (DATA for an open socket, WIFI_STATE for a status display), handle it and continue waiting.
4. Otherwise skip the payload and continue waiting.

The slot ROM follows this pattern while it waits for `$D1`: an event that arrives ahead of the block reply is skipped by its length and the wait continues. The `cogslib` helpers do the same with `cogs_recv_frame`, and `cogs_poll_event` returns the events a caller wants to observe.

CHAPTER 6

The Network Disk

The card presents a SmartPort block device whose blocks come from a disk image on an HTTP or HTTPS server. No software is installed on the Apple II: the slot ROM turns the operating system's block calls into VOL_READ and VOL_WRITE frames, and the card fetches or stores the bytes over the network. This chapter describes the slot ROM, the boot sequence, block fetching and caching, the mount table, writable volumes, and the sources a volume can come from.

6.1 The slot ROM

The ROM occupies the `$Cn00` page and the `$C800` to `$CFFF` shared window. The `$Cn00` page contains no absolute references to itself, so one image serves every slot; the `$C800` body is at a fixed address and is served by the card whenever the card holds the window.

Table 6-1. Slot ROM signature bytes

Address	Value	Meaning
`$Cn01`	`$20`	ProDOS block device and SmartPort signature
`$Cn03`	`$00`	
`$Cn05`	`$03`	
`$Cn07`	`$00`	SmartPort present
`$Cn0A`		Alias of the ProDOS entry, the IIGS internal-SmartPort convention
`$Cn0D`		Alias of the SmartPort entry
`$Cn80`		ProDOS block device entry
`$Cn83`		SmartPort entry
`$CnFB`	`$80`	SmartPort flags: extended calls supported
`$CnFE`	`$17`	ProDOS characteristics: two drives, status, read, write
`$CnFF`	`$80`	ProDOS entry offset

Software that reads `$CnFF` finds the entry at `$Cn80`. Software that assumes the IIGS internal drive convention and calls `$Cn0A` or `$Cn0D` directly reaches the same dispatch through the aliases.

6.1.1 ProDOS block device entry

The `$Cn80` entry takes the standard ProDOS parameters in zero page: command at `$42` (0 STATUS, 1 READ, 2 WRITE, 3 FORMAT), unit at `$43` (drive in bit 7, slot in bits 4 to 6), buffer at `$44/$45`, block at `$46/$47`. The driver saves and restores `$42` to `$47` and touches no other zero page, because a ProDOS boot loader keeps live state at `$48` to `$4F`. The two ProDOS drives map to units 1 and 2. Errors return with carry set and a code in A: `$27` I/O error, `$2B` write protected, `$28` no device.

6.1.2 SmartPort entry

The `$Cn83` entry takes the standard inline command byte and parameter list pointer. Standard and extended calls are both accepted; extended calls carry 24-bit buffer addresses and are the only path that executes 65816 instructions. All other code in the ROM is NMOS 6502, so the same image runs in a II Plus, an enhanced IIe, and a IIGS.

STATUS code 0 on unit 0 returns the device count and seven zero bytes, and from firmware 1.30 also sends the card SP_SCAN with payload 0, the boot marker (see "Pending eject" below). STATUS code 0 on a unit returns the general status byte and a 24-bit block count. From firmware 1.31 every per-unit STATUS also sends SP_SCAN with payload 1 + status code, which is how the card tells GS/OS's driver generation (unit 0, then a DIB) from ProDOS 8′s loader (unit 0, then code 0) and clears its CDA latch only for the former (Chapter 4, SP_SCAN). STATUS code 3 returns the Device Information Block:

Table 6-2. Device Information Block

Field	Value
General status	`$F0` block device, online, read and write allowed; `$F4` when the mount is read-only (write-protect set, write-allowed still set); `$E0` present but not online for an unconfigured unit. Bit 0 is set once when the unit's disk-switched flag is pending.
Block count	From VOL_STAT or the unit status view
ID string	`COGS.NETWORK`
Device type	`$02`, hard disk
Device subtype	`$C0`, extended calls and disk-switch capable
Firmware version	`$1000`

Units are removable. An unconfigured unit reports present but offline (`$E0`) so the operating system shows an empty drive instead of failing the slot scan.

Write-allowed (bit 6) is always set, on empty and read-only units too. GS/OS builds a unit's device characteristics once, at boot, from this byte (the code 3 status), and a unit that reported write-allowed clear then stays "not writable" for the whole session: the Finder padlocks every volume later mounted there and GS/OS refuses writes at the device level with invalid access. Read-only is signalled by the write-protect bit alone, which is what the Finder lock and the FST's write-protected error come from.

Interrupts are masked for the whole of every driver call. The driver runs in the shared `$C800` window; an interrupt handler that touched another slot's `$CnXX` space or `$CFFF` mid-call would deselect the ROM under the executing code.

6.1.3 Unit count

The ROM presents eight units. In slot 3 it presents four: GS/OS does not run native multi-unit SmartPort through the shared slot 3 ROM space and falls back to the ProDOS two-drive plus phantom-slot path, which tops out at four units. The count is reported to the card in the CONFIG `units` byte so the card's drive pages clamp to match.

6.2 Boot sequence

The `$Cn00` page is also the boot entry. When the Apple II selects the slot at power-on:

1. The ROM clears the CDA latches in the text-page screen holes for its slot and claims `$C800`.
2. It samples the keyboard for about 0.4 seconds. M or C enters the config terminal (Chapter 7). A digit 1 to 8 selects that drive as drive 1 for this session. A key held from power-on registers without waiting for auto-repeat; a keystroke left in the latch from before the ROM ran is discarded.

3. It sends BOOT_PICK with the digit or 0. If the reply asks for the boot chooser, it enters the config terminal, which opens on that page.
4. It reads block 0 of unit 1 into `$0800` with VOL_READ, retrying while the card reports the link is not yet up, and continues to sample the keyboard during the wait.
5. It jumps to `$0801` with X holding the register base (`$80` plus the slot times 16), the standard ProDOS boot convention.

A ProDOS boot block then pulls the rest of the operating system through the `$CnFF` driver entry. A disk image whose block 0 bypasses the block device (a copy-protected or custom loader that drives floppy hardware directly) reads block 0 and never returns; such images do not boot from the card.

6.3 Block fetch and caching

A VOL_READ that misses every cache costs one HTTP range request. Three mechanisms keep most reads off the network.

Read-ahead window. A miss fetches 32 blocks (16 KB) starting at the requested block in one ranged GET, and the following sequential reads are served from that window. The window is allocated at the first fetch and freed on close. At most two mounts hold a window at once; when a third needs one, the least recently used idle window is flushed to the flash cache and released, and that mount refetches on its next miss. A window with a fetch on the wire is never taken. To a host that does not redirect, the connection is kept alive and parked between windows, so one TLS handshake serves a whole boot. A redirecting host (a CDN front) is reconnected per window because the target can change.

Flash warm cache. Every block fetched is also written into a persistent cache in the card's flash: 765 lines of 7 blocks (3584 bytes) in 4 KB sectors, about 2.6 MiB. A line is keyed by a hash of the URL and a content tag (the origin's ETag or Last-Modified header when it sends one, otherwise the file size). A changed image fails the tag and is refetched; a same-size replacement with no validator can serve stale lines until the cache is wiped from the card menu. The cache is read-only with respect to the network: volume writes never land in it, so flash erase count grows only with new blocks seen. Every sector is self-describing and the directory is rebuilt by scanning at power-on. A cold boot from an image whose working set is cached completes with no network reads.

Boot manifest. The card records which cache lines a boot touches, as a short list of line ranges per image, and on the next cold boot of the same image streams those ranges into the flash cache ahead of the host's requests. A wrong or stale manifest costs bandwidth, never correctness.

6.3.1 Delivery to the host

A 512-byte block reaches the host by one of two paths.

FIFO. The `$D1` reply is 9 header bytes plus 512 data bytes in the card-to-host FIFO. The ROM reads the header with STATUS-gated pops, waits for BULK to report 512 bytes queued, then pops DATA 512 times without polling. This is the path on 6502 and 65C02 hosts.

Block window. On a 65816 host the ROM sets VOLRD_WIN in the request. The card parks the 512 bytes in a fourth `$C800` view and queues only the 9-byte header. The ROM pulses CONTROL bit 7 to map the view, copies `$C800` to `$C9FF` to the caller's buffer with one MVN (the destination bank is in the operand, so extended SmartPort calls land directly in the caller's buffer), and pulses CONTROL bit 6 to restore the driver body. A buffer whose 512 bytes would cross a bank boundary uses the FIFO path instead, because MVN wraps within a bank.

Writes use the window in the other direction: the ROM maps the view, MVNs the caller's buffer to `$C800`, restores the body, and sends the 6-byte VOL_WRITE header with VOLWR_WIN. The card counts the

stores it saw while the view was mapped and takes the block only at 512 or more; otherwise it answers WIN_SHORT and the ROM resends with the data in the frame.

6.3.2 Error mapping

Table 6-3. Result code mapping

Card result	ProDOS / SmartPort error
0 OK	Carry clear
11 UNSUPPORTED on a write, or VOLF_RO set	`$2B` write protected
Unit past the effective count	`$28` no device connected
Any other nonzero result, or a timed-out wait	`$27` I/O error

Every wait in the ROM that could block is bounded. A missing reply is an I/O error to the operating system, not a hang. A block read is retried up to eight times, each retry beginning with a flush of the card-to-host FIFO and a fresh request; the card serves the repeat from cache.

6.4 The mount table and units

The card keeps one URL per unit in flash. Unit 1 is the boot volume and is the same entry VOL_URL_SET and VOL_URL_GET address; units 2 to 8 are set with VOL_MOUNT_SET. An empty entry is an unconfigured unit.

6.5 The recent-disk list

Twelve last-used volume URLs, newest first. Browse hub R (under Featured) and Disk Drives R read the same list. Every successful VOL_URL_SET or VOL_MOUNT_SET writes it, and so does Browse Return or a digit mount. There is no opcode for the list.

Each row holds a display name, the URL, a boot flag (the B mark), a your-source flag (ys), and a bundle tag: none, TR (Total Replay), MC (a MECC volume), or HD (any other collection image), firmware 1.26. Browse `recent_remember_ex` stores the catalog title, the flags and the tag it painted. The opcode path (`recent_remember`) stores the URL with boot set, ys clear and no tag, except a URL whose path looks like Total Replay is tagged TR and a MECC volume filename is tagged MC; nothing infers HD from a URL, an HD row is only ever a child row Browse remembered as one. A second remember of the same URL with no title keeps the stored name, flags and tag and moves the row to the front.

The Pico keeps the list in one flash sector next to the crash log. Magic is `CGR2`, with a flags byte per slot (bit 0 boot, bit 1 TR, bit 2 ys, bit 3 MC, bit 4 HD). A leftover `CGR1` sector loads as boot set, everything else clear; rows written before 1.26 have the MC and HD bits clear until they are mounted again. The emulator writes `~/.cogs_emu_recent.json` (or `COGS_EMU_RECENT`) with `name`, `url`, `boot`, `tr`, `mc`, `hd`, and `ys` on each item. A JSON file that omits the flags loads as boot set, the rest clear. Factory Reset clears the list. The first boot after the list is empty copies the current mount table in.

On a Browse list the marks sit size (cols 26-29), then 8b or Dn (31-32), then the bundle tag (34-35), then V (37), then B (39); a row reads `800K 8b TR B`, `32M D1 MC B` or `800K HD B`. The tag follows the catalog row's `in` parent name: a parent starting `Total` paints TR, one starting `MECC` paints MC, any other parent paints HD, and a row with no `in` has no tag. The picked screen names the parent and adds `Then pick from its menu:` with the title; that hand-off keys off the parent name, not the tag. Recent paints from the stored tag, never from the hand-off marker. On Recent, X drops one row and A clears the list. Both refuse while that image (or any listed image, for A) is still mounted. A ys row mounts only when Image Source is a custom catalog whose host matches the URL. Otherwise the menu says `Your source changed or no longer available.` The check is the host string, not a live HTTP probe.

A unit is mounted on first touch: the first VOL_READ or VOL_STAT on a handle with no open volume opens the stored URL. Setting a mounted unit's URL takes effect at once. The card drops the open volume, raises the unit's disk-switched flag, and the next read mounts the new image; the ROM reports the flag in the SmartPort status byte and GS/OS remounts the volume on the running desktop.

BOOT_PICK with a digit swaps that unit into position 1 for the session without changing the stored table, so a warm restart without a key returns to the stored order.

6.5.1 Two-phase remount

GS/OS binds a volume name to the device it was mounted on and only revises that binding when a STATUS poll reports the device empty or disk-switched. A unit that changes contents while GS/OS is not polling (the desk accessory has the machine, or the change came from the card's own housekeeping) would otherwise be seen for the first time already holding the new image, and GS/OS would treat the new name as a second disk on a device it still maps to the old name. Emptying and refilling several units in one CDA visit produced the "two disks named X, rename or eject" dialog and a Finder crash that way.

From firmware 1.26 the card holds such a unit empty until GS/OS has looked once. When an open volume is dropped behind GS/OS's back (VOL_MOUNT_SET over a mounted unit, the CDA's X, the share-gone dismount below), the unit enters a held state: the unit status view publishes not-valid so the ROM takes the frame path, and the first VOL_STAT, VOL_READ or VOL_WRITE for that handle answers 16, VOL_NOMOUNT, clears the hold, and raises the disk-switched flag again. The ROM maps that reply to offline (`$E0`, 0 blocks), GS/OS drops the old volume, and the next poll finds the unit online, disk-switched, and holding the new image (or still empty). The mount probe for the new image runs during the hold, so the new volume is usually warm by the time GS/OS asks for it.

The hold exists only while a GS/OS desktop is live. The card learns that from CDA_INSTALL (the Desk Manager fetched the installer) and from CONFIG with the CDA source byte, and forgets it on BOOT_PICK, the boot-ROM CONFIG entry, "boot it now", and a bus reset. At boot the ROM's own probes open units too, and a hold there would make the boot-time STATUS report the unit empty once. VOL_EJECT never sets a hold: GS/OS asked for that eject and knows the unit is empty.

6.5.2 Pending eject

GS/OS sends the same SmartPort CONTROL 4 (Eject) for two things that want opposite results. A Finder Eject means "take this drive away", and the card removes the stored mount, in the CDA and in Image Manager. Shut Down (`dd_full_shutdn`, dispatcher call 3) ejects every device whose DIB says removable media, the card's eight units included, and there the mounts must survive. Firmware 1.23 to 1.28 cleared the stored URL on both, so every Shut Down emptied the card.

From 1.29 VOL_EJECT drops the live volume and marks the unit's eject pending; the stored URL is untouched. The card decides by the next frame it receives, not by a timer. A Finder Eject is followed at once by GS/OS's own STATUS of that unit (the poll that used to bring the volume straight back before 1.23), so the first non-eject frame after an eject is treated as the session continuing and the pending mounts are removed then, exactly the old way (`vol_mount_set` with an empty URL, remount, GS/OS-observed). After Shut Down's ejects nothing reaches the card: the dispatcher's driver shutdown is a no-op on the wire for a class 2 device, and the notify queue ran before the ejects. The next frame is the next boot's, and a boot lifts every pending eject with the mounts kept.

Three things count as a boot. A bus RESET (power switch, Control-Reset, MAME's reset) resets the session and clears the pending set before any frame. BOOT_PICK, the slot ROM's first frame when the card is the boot device, and the boot-time CONFIG do the same. SP_SCAN (firmware 1.30) covers the remaining case: GS/OS's Restart from the "You may now switch off" screen is a software reset (`restart_system`: `sta $C068`, `jmp $FA62`) with no RESET on the bus, so a card in a slot other than the boot slot used to see the new boot's first per-unit STATUS as a Finder poll and remove the mounts. The ROM

now sends SP_SCAN from the SmartPort unit-0 STATUS, the device-count call GS/OS's driver generation (`sp_unit0`) and ProDOS 8′s slot scan make once, before any per-unit call, and never again on a running desktop. Restart-with-eject (OSShutdown flag bit 2) lands on the same side: the reboot lifts it.

The pending flag is per unit, never persisted, cleared for one unit when its mount changes (including that removal) and for all units by a boot. While it is set the unit reads offline to the host whatever the stored table says. A false boot marker (a utility issuing a unit-0 STATUS inside the fraction of a second between a Finder Eject and GS/OS's poll) would keep a mount the user meant to remove, the pre-1.23 behaviour; no software is known to do that.

6.5.3 Share write-protect poll and dismount

A share that sends `X-CoGS-Poll: N` on its image responses asks the card to re-fetch 64 bytes of each mounted image from that share every N seconds (2 to 60) while the whole bus has been idle for 3 seconds and no other card traffic is live. Each poll response carries `X-CoGS-Writable: 0|1`; a change latches or clears the unit's read-only flag and raises disk-switched, so the Finder's next poll redraws the lock. Hosted origins send neither header and are never polled. A poll that has not answered in 8 seconds is dropped and its connection aborted, so a share that vanished with a warm keep-alive connection cannot stall later polls.

Three consecutive polls unanswered (connection refused, timed out, or a non-2xx status) while the card's own link is up mean the share is gone. The card then clears the unit's stored URL and drops the volume, the same as VOL_EJECT, and the two-phase hold makes GS/OS's next STATUS see the unit empty. Failures while the card's link is down do not count. Bringing the share back does not remount; the user mounts again from Recent or Your source. The Image Manager sends `X-CoGS-Poll: 5`.

6.6 Writable volumes

VOL_WRITE stores a block with a ranged HTTP PUT to the origin and invalidates the block in the read-ahead window and the flash cache. Whether an origin accepts writes cannot be known in advance; the first write the origin rejects with 403, 405, or 501 latches the mount read-only. From then on writes fail without network traffic, VOL_STAT reports VOLF_RO, and the ROM reports write protected, so the operating system sees a locked disk rather than repeated I/O errors. Re-pointing the unit clears the latch.

The a2cogs catalog, GitHub, and the Internet Archive are read-only origins. The CoGS Image Manager share and `cogs_share.py --writable` accept PUTs.

6.7 Disk image formats and sources

Table 6-4. Disk image formats

Extension	Handling
`.2mg`	The 2IMG header is parsed at VOL_OPEN; the data offset and block count come from it, and the count is clamped to the blocks the file holds.
`.po`, `.hdv`	Raw ProDOS-order blocks from byte 0. The block count is the file size divided by 512.

Images must be flat ProDOS-order files on an HTTP or HTTPS server that honors range requests with a 206 response; DOS-order and nibble images are not block devices. The card follows up to three redirects when opening and fetching, so an Internet Archive `/download/` URL or a GitHub release asset works as a source. The Browse pages list images from a catalog at the base URL stored with CAT_URL_SET (the a2cogs catalog by default); a LAN share serves the same catalog shape for a folder of images. Appendix B gives the formats, the catalog shape, and the share endpoints.

CHAPTER 7

Desk Accessory and Tool Injection

On an Apple IIGS the card supplies its own classic desk accessory and its audio tool sets to the running system without any file on the boot volume. This chapter describes the `$C800` overlays the card serves, the two-stage installer that places **CoGS Config** in the Desk Accessories menu, the `tool://` stream that serves relocated tool sets, and the rules that keep both safe under a live GS/OS.

7.1 The `$C800` views

The card serves four different 2 KB images in the shared `$C800` to `$CFFF` window, selected by CONTROL pulses. The Apple II's claim on the window follows the normal slot rule; the pulses change only what the card returns for IOSTRB reads.

Table 7-1. `$C800` views

View	Mapped by	Content
Driver body	CONTROL bit 6 (default)	The SmartPort and ProDOS block driver (Chapter 6)
Config overlay	CONTROL bit 4	The boot-time config terminal loop and the IIGS clock write
Installer	CONTROL bit 5	Stage 1 of the desk accessory installer
Block window	CONTROL bit 7	512 bytes of block data at `$C800` to `$C9FF`

The driver body keeps 7-byte entry pads at `$C800` (CONFIG_RUN) and `$C807` (CLK_BOOT) that pulse the config overlay on; the overlay assembles matching bytes at the same addresses so control passes cleanly whichever view the card is serving when the CPU fetches. An overlay never calls into the driver body, which is not visible while the overlay is mapped. Code in any view never executes from `$CFFE` or `$CFFF`: the processor's dummy fetch of the next byte would read `$CFFF` and release the card's claim on the window.

7.2 The config terminal

The card renders its menus and the Apple II is a display and keyboard. The host loop is about 200 bytes: send CONFIG with a key byte, read the 961-byte reply, copy 960 screen codes to the text page, wait for a key or, when the reply's POLL flag is set, send TICK again. Every face that shows the card menu (the slot ROM at boot, the card-resident desk accessory, the disk-based Config Console, the 8-bit console) is this loop. The request's optional bytes tell the card what the host can do (unit count, lowercase, slot, clock, entry source), and the reply flags ask the host for the few actions only it can perform: write the system clock, run a local audio player, stop touching the card before a firmware restart.

7.3 The desk accessory installer

CoGS Config is a classic desk accessory installed with the Desk Manager's InstallCDA call from code and data the card supplies. Installation is triggered from inside the block driver, deferred to a safe context by the Scheduler, and completed by a position-independent image the card streams into a locked handle.

7.3.1 Trigger and gate

Every SmartPort STATUS call that reaches the driver while the desk accessory is not yet installed runs stage 1 in the installer overlay. Stage 1 checks, in order, and returns without effect unless all hold:

1. `$E100BC` (the OS_KIND byte) is `$01`: GS/OS is the running operating system. `$00` is ProDOS 8 or pre-boot; `$FF` is the GS/OS boot transition, during which the toolbox is not ready on every ROM.
2. The Tool Locator and Desk Manager function pointer tables point into RAM, not ROM: the RAM tool sets have loaded.
3. The Window Manager's work area pointer is nonzero: the desktop is up.
4. Twelve consecutive STATUS calls have arrived with VOLIO unchanged: the calls are the Finder's idle poll, not the block traffic of an application launch.

Once the gate holds, stage 1 latches "dispatched" in the text-page screen hole `$0478` plus slot (`$A5`) and in the CDA register (`$A5`), and the driver adds a Scheduler task from the `$Cn00` page. The Scheduler runs the task when the system busy flag next falls to zero, which is the legal context for memory-moving toolbox calls; nothing is allocated from inside the driver call itself.

7.3.2 Stage 1: the overlay

Stage 1 is 831 bytes served in the installer view. It is never copied to RAM; the card serves it in place, and the only RAM it touches is the caller's stack. Its Scheduler task, entered at `$C803`:

1. Obtains a memory ID of the desk accessory class (`$5000`) with GetNewID. Desk accessory class IDs survive application shutdown; an application ID would be purged with the application.
2. Allocates a locked, fixed handle that may not cross a bank, of the stage 2 size.
3. Requests CDA_INSTALL with selector 1 and copies the reply into the handle, reading the FIFO with 8-bit index registers.
4. Writes the slot (offset 12) and the memory ID (offset 14) into the stage 2 header.
5. Calls the stage 2 install entry at offset 0.

7.3.3 Stage 2: the resident image

Stage 2 is 15 135 bytes, assembled at origin 0 and position-independent: internal calls use PER and BRL, and data references go through a base computed at run time. It stays in its locked handle for the life of the desk accessory.

Table 7-2. Stage 2 header

Offset	Field
0	Install entry (JSL)
8	Reserved
12	Slot, 1 to 7
14	Memory ID
16	Installed flag

The install entry calls InstallCDA with the desk accessory header inside the image, sets the installed flag, and latches success in `$04F8` plus slot (`$5A`). The same image contains the desk accessory itself: the config terminal loop, the audio player, and the picture viewer. The desk accessory runs with a private direct

page and a 4 KB stack block it allocates on entry; when audio is playing in the background the bottom half of that block serves the heartbeat task and the top half serves the reopened desk accessory page.

7.3.4 Latches

Table 7-3. Installer latches

Location	Value	Meaning
`$0478` + slot	`$A5`	Dispatched this session; stage 1 does not re-queue
`$04F8` + slot	`$5A`	InstallCDA succeeded
CDA register (`$C0n8`)	`$A5`	Card copy of the dispatched latch

A 40- or 80-column screen clear wipes `$0400` to `$07FF`, so the screen holes alone are not reliable. Stage 1 restores them from the CDA register when it finds the card copy set, and does not queue again. The slot ROM clears both holes on every boot from the card; the card clears its copy on slot RESET and on a soft reset, both of which mean a fresh Desk Manager.

7.4 Card-served tool sets

The audio player in the desk accessory requires two tool sets that are not part of the system software. The card carries both and serves them through the byte stream interface.

Table 7-4. Card-served tool sets

Tool set	Content	Image size
226	PCM streaming player for the Ensoniq DOC	39 621 bytes
219	SoundSmith music player 1.1.3	5 569 bytes

7.4.1 The `tool:// stream source`

```
tool://NNN?base=XXXXXX
```

NNN is the tool set number and XXXXXX is exactly six hexadecimal digits, the address the client has allocated for the image, without a dollar sign. STREAM_OPEN with this URL opens handle 1 and needs no network; it coexists with the network stream on handle 0. The card holds each tool set as a flat image at origin 0 plus a relocation table, applies the relocations for the requested base, and serves the result exactly as the System Loader would have left it at that address. The image begins with the function pointer table, so the client passes the block address directly to SetTSPtr.

STREAM_READ on handle 1 returns min(max_len, remaining) bytes on every call and raises END on the read that delivers the last byte; `level` is 255 while data remains. STREAM_CLOSE frees the handle. A second `tool://` open while one is open replaces it. A tool set number the card does not carry returns UNSUPPORTED; a missing or malformed `base` returns BAD_FRAME.

7.4.2 Client requirements

The destination block must be locked, fixed, and must not cross a bank boundary, matching the load segment's BANKSIZE attribute. The client sequence is:

1. NewHandle for the image size with those attributes.
2. STREAM_OPEN `tool://NNN?base=<address>`.
3. STREAM_READ until END, copying into the block.
4. STREAM_CLOSE.
5. SetTSPtr with the tool set number and the block address.

6. The tool set's own StartUp.

7.5 Residency and precedence

The card copy is primary. The desk accessory loads a tool set from the card first and falls back to LoadOneTool from the boot volume only when the injection fails for a reason other than memory. A tool set installed by either path stays in the Tool Pointer Table until restart, and the desk accessory records which sets are present so a later play does not install over a loader-installed copy. An allocation failure at any step (the tool image, the audio ring, the private stack) is reported to the card as PLAYMEM so the card's error page names memory rather than a missing tool.

Tool set 226 is copyright Petar Puskarich and is distributed with CoGS under permission. Tool set 219 is Brutal Deluxe freeware, redistributed unmodified. Both ship only inside the card firmware and the emulator; neither is republished on its own.

CHAPTER 8

Bus Timing Rules

The Apple II slot bus and the 6502-family processors that drive it have properties that host code must respect when it touches a FIFO register. Chapter 2 states the rules; this chapter shows the bus cycles behind each one, so that code which must break a rule for a good reason can do so knowingly.

8.1 The read deadline

A slot I/O cycle runs at the 1 MHz bus rate on every Apple II. The select line for the card's region (/DEVSEL for the registers, /IOSEL for `$Cn00`, /IOSTRB for `$C800`) falls early in the cycle and the processor latches the data bus at the end of PHI0. The card has about 500 ns from the fall of the select line to drive valid data.

The card meets this with state machines that capture the address and drive a pre-staged byte with no software in the path. Every readable register value is computed before the cycle that reads it: STATUS is maintained as the FIFO heads and tails move, RXLO/RXHI are snapshots, and the byte DATA will return next is already latched. The consequences for the register design are that no register read can run code, take a lock, or wait, and that STATUS carries only flags that cost nothing to keep current. A host cannot cause a bus timeout by reading any register at any rate, including 512 back-to-back DATA pops.

An accelerated IIGS still runs slot I/O cycles at 1 MHz. Register timing is identical on every host; only the instruction time between register accesses changes.

8.2 Indexed-store dead cycles

Every 6502-family processor performs a read of the target address in the cycle before an indexed store writes it.

Table 8-1. `sta abs,X` cycle by cycle

Cycle	Address bus	Data bus	R/W
1	PC	opcode `$9D`	read
2	PC+1	address low	read
3	PC+2	address high	read
4	address + X, high byte not yet carried	dummy	read
5	address + X	data	write

When the index does not cross a page, the cycle 4 address is the target address itself, and the register base `$C080 + n*16` never crosses a page for any slot. On DATA that read is a pop from the card-to-host FIFO whose data the processor discards. Uncompensated, every command byte a host sent with `sta DATA,x` would pop one byte of whatever reply or event the card had already queued, and the next frame parse would fail. Processors that keep the dummy cycle internal (some accelerator cores) do not show the

effect, and an emulator that does not model dummy cycles does not either, so code with this fault can pass on both and fail on a stock machine.

The card compensates on the write path, which has none of the read path's deadline pressure: when a DATA write arrives within 2 µs of a DATA read, the read was the store's own dead cycle, and the card restores the popped byte to the head of the FIFO (or, if the FIFO was empty, cancels the underrun count). The dummy cycle is about 1 µs before the write; a genuine read-then-write of DATA by host code takes at least 5 µs at the bus rate, so the window cannot misfire on legitimate traffic. `sta DATA,x` is therefore a correct way to write a command byte on every host, and is what the slot ROM uses.

The compensation restores exactly one byte per store. Read-modify-write instructions on DATA (`inc abs,X`, `asl abs,X`) read the target twice and are not covered, and have no use on a FIFO register. The wide-window registers at +B and +C ignore writes and their reads are guarded (below), so an indexed store aimed at them, including the spill of a 16-bit store to SYNC, is harmless.

8.3 The 16-bit index extra cycle

The 65816 in native mode with a 16-bit index register adds a cycle to `lda abs,X` that the 8-bit-index form does not have.

Table 8-2. `lda abs,X` with a 16-bit index register

Cycle	Address bus	Data bus	R/W
1	PC	opcode `$BD`	read
2	PC+1	address low	read
3	PC+2	address high	read
3a	address high, (address low + X low) with no carry	dummy	read
4	address + X	data	read

Cycle 3a occurs when the index register is 16 bits wide (x = 0 in the status register), or when the page boundary is crossed with an 8-bit index. With x = 0 it occurs on every execution. When the index does not cross a page the dummy address equals the target, so a `lda DATA,x` pops two bytes and returns the second. The corruption is silent: the host sees a stream shifted by one byte per read.

The rule: never read a card register with `abs,X` or `abs,Y` while the index register is 16 bits wide. The safe forms are:

- `sep #$10` (or `sep #$30`) before the register access and `rep` after, so the index is 8 bits and cycle 3a does not occur.
- Direct-page indirect long addressing, `lda [dp]` or `lda [dp],y`, which has no dummy read of the target.
- Absolute long addressing with the register address folded into the instruction, `lda $E0C0n0`.

Emulation-mode code and all 8-bit hosts are unaffected because the index is always 8 bits. The 65816 in some accelerator cores does not emit cycle 3a, so code with this fault can run correctly on one machine and fail on another. Code that reads the FIFO with a 16-bit index may appear to work on such a machine and must not be taken as proof.

8.4 Accelerators and re-issued I/O cycles

An accelerator card or a fast IIGS core slows to the 1 MHz bus rate for slot I/O, and some implementations do this by re-running the I/O cycle after the processor has already latched the bus, or by presenting the address for more than one bus cycle. To the card this looks like two reads of the same register in consecutive cycles.

A re-run read of STATUS, RXLO, RXHI, BULK, or any other side-effect-free register is harmless. A re-run read of DATA would pop twice. A BULK-gated burst drain (512 consecutive DATA pops) is nonetheless correct on every host that has been measured, because a re-run cycle presents the same address and data and the processor discards the duplicate; the card sees one pop per host read. Code that relies on the burst drain should confine it to the case BULK guarantees (512 or more bytes already queued) so that a pop can never block.

The wide window has an explicit guard for this case. A legitimate wide drain reads +B, +C, +B, +C, alternating. If the card sees two consecutive pops at the same offset it returns the same byte again without popping. A re-run of either half of a 16-bit read is therefore absorbed. The guard is also why single odd bytes must never be popped through +B or +C: two single pops at the same offset would be treated as a re-run and the second would return a stale byte. Odd trailing bytes go through DATA.

8.5 Interrupts during a drain

Two things can go wrong when an interrupt is taken between FIFO accesses.

Code running from `$C800`. The block driver executes from the shared ROM window. An interrupt handler that reads any other slot's `$CnXX` space, or `$CFFF`, transfers the window to that slot or releases it. When the handler returns the processor fetches its next instruction from whatever now occupies `$C800`. The driver masks interrupts for the whole of every call and does not re-enable them before returning; the operating system restores its own interrupt state.

A handler that touches the card. A handler that pops DATA while foreground code is mid-frame steals bytes from the foreground's frame, and the foreground then waits for bytes that have already left. The rule is that one context owns the card at a time. Foreground code that performs a frame round trip while an interrupt-driven client (the background audio heartbeat) is active brackets the round trip with `sei` and `cli`; the heartbeat task itself never enables interrupts and finishes well inside its time budget.

The Ensoniq DOC interrupt service on the IIGS runs about 5 ms of every 16.7 ms frame while a tool set 226 stream is playing. A heartbeat task that also drains the FIFO must fit its whole entry, worst case, inside half a frame with that service present; an interrupt chain that overruns the frame does not degrade gracefully but locks the machine in interrupt context. The card's own heartbeat reads at most 448 bytes per tick.

8.6 Slot RESET

The card treats the slot RESET line as a hard reset of the bus interface: both FIFOs are flushed, the frame parser is reset, the CDA latch is cleared, and the `$C800` view returns to the driver body. Settings in flash, open network connections, mounted volumes, and the flash cache are unaffected; pending ejects are lifted (Chapter 6, "Pending eject"). A config terminal session abandoned by Control-Reset is discarded by the card when the next command arrives. GS/OS's Restart from the shutdown screen is not a RESET on the bus (it is `sta $C068`, `jmp $FA62`), which is why the boot marker SP_SCAN exists: its payload-0 form lifts pending ejects (1.30), and a unit-0 marker followed by a DIB marker clears the CDA latch the way RESET does (1.31), so the desk accessory installs into the new Desk Manager.

RESET is taken by the card as an interrupt; a RESET edge that arrives while the card is programming its own flash is deferred until the flash operation completes and then acted on. Host code sees no difference except that the first register read after RESET may be delayed by a few milliseconds in that case.

The card's processor can come out of a power dip a fraction of a second before the host's RESET line settles, so the card may see the tail of the host's reset as a very short pulse, with the falling and rising edges latched into one interrupt. Firmware 1.27 and later reads the pin level in that case and treats the line as released if it is high, so the bus interface is enabled in time for the slot scan. The host-to-card

queue is emptied only up to the point where RESET fell; bytes the host writes after the edge are kept, so a slot ROM that starts talking to the card while the card is still finishing its own start-up (about two seconds after power on a card with the wired port) gets its first frame answered rather than dropped. Each RESET edge is recorded in the boot log with its time, the pin level and the state of the bus engine.

CHAPTER 9

Programming the Card

This chapter shows how to talk to the card from your own programs: detecting it, sending a command and reading its reply, draining bulk data, fetching over HTTPS, extracting one value from JSON, using the hash and random services, and mounting a disk image. Examples are given in 65816 assembly, in ORCA/C using the `cogslib` library, and in cc65 C for 8-bit hosts. The assembly is ca65 syntax.

Throughout, `BASE` is `$C080 + n*16` for slot n. The register offsets from Chapter 2 are:

```
DATA    = $C080         ; + n*16
STATUS  = $C081
RXLO    = $C082
RXHI    = $C083
CONTROL = $C084
ID      = $C085
VERSION = $C086
BULK    = $C087
```

With X holding `n*16`, `lda DATA,x` addresses the slot's DATA register. On the IIGS in native mode the data bank must be `$E0` (or bank 0 with I/O shadowing in effect) and the index register must be 8 bits wide (Chapter 8).

9.1 Detecting the card

Three reads, no timing dependence. ID reads `$A2`, VERSION is nonzero, and STATUS is not `$FF`.

```
; entry: X = slot * 16. exit: carry set if a card is present
detect: lda     ID,x
        cmp     #$A2
        bne     @no
        lda     VERSION,x
        beq     @no
        lda     STATUS,x
        cmp     #$FF
        beq     @no
        sec
        rts
@no:    clc
        rts
```

To find the slot, run the probe for X = `$10`, `$20`, … `$70`. A slot with no card returns floating-bus values, which fail the ID compare.

In ORCA/C with `cogslib`:

```
#include "cogs.h"
#include "cogs_io_slot.h"

static cogs      card;
static cogs_io   io;
static cogs_slot slot;

int slotno = cogs_io_slot_find(&io, &slot);   /* 1..7, or 0 if none */
if (slotno == 0) { /* no card */ }
cogs_init(&card, &io);
```

The IIGS Control Panel must have the card's slot set to **Your Card**; otherwise /DEVSEL never reaches the slot and the probe fails.

9.2 Sending a command and reading the reply

A command is the opcode, two length bytes, and the payload, each written to DATA after TX_READY (STATUS bit 6) is set. A reply is read the same way in reverse after RX_AVAIL (STATUS bit 7) is set. The example sends PING and reads the four-byte reply.

```
; entry: X = slot * 16, 8-bit A and X
ping:   lda     #$00            ; PING
        jsr     wr
        lda     #$00            ; LEN lo
        jsr     wr
        lda     #$00            ; LEN hi
        jsr     wr
        jsr     rd              ; opcode, expect $80
        cmp     #$80
        bne     ping_err
        jsr     rd              ; LEN lo = 4
        sta     len
        jsr     rd              ; LEN hi = 0
        ldy     #0
@pay:   jsr     rd
        sta     reply,y
        iny
        cpy     len
        bne     @pay
        rts

wr:     bit     STATUS,x        ; bit 6 -> V
        bvc     wr
        sta     DATA,x
        rts

rd:     lda     STATUS,x        ; bit 7 -> N
        bpl     rd
        lda     DATA,x
        rts
```

`sta DATA,x` is safe; the card compensates for the indexed store's dummy read (Chapter 8). `rd` as written waits forever; production code counts iterations and gives up, as the slot ROM does.

In `cogslib`, every command is one call:

```
cogs_ping_reply p;
if (cogs_ping(&card, &p) == COGS_OK) {
```

```
    printf("firmware %u.%02u proto %u caps %02X\n",
           p.fw_major, p.fw_minor, p.proto_ver, p.caps);
}
```

A command with no helper is sent with the framing primitives:

```
static cogs_u8 buf[64];
cogs_u8 op; cogs_u16 len;
buf[0] = 5;                                  /* RANDOM: count */
cogs_send_frame(&card, 0x31, buf, 1);
cogs_recv_frame(&card, &op, buf, sizeof buf, &len);   /* op == 0xB1, len == 5 */
```

`cogs_recv_frame` returns the next frame whatever its opcode. A caller that may receive events checks `op` and loops.

9.3 Draining bulk data

Byte-at-a-time reading through STATUS runs at about 22 to 25 KB/s. Two faster patterns exist for large replies.

Burst. When the remaining payload is at least 512 bytes, wait for BULK to be nonzero, then pop DATA 512 times with no poll.

```
; 512 bytes from DATA to (ptr), 8-bit A/X/Y
burst:  lda     BULK,x
        beq     burst           ; wait for 512 queued (bounded in real code)
        ldy     #0
@lo:    lda     DATA,x
        sta     (ptr),y
        iny
        bne     @lo
        inc     ptr+1
@hi:    lda     DATA,x
        sta     (ptr),y
        iny
        bne     @hi
        rts
```

Wide. On a 65816 with VERSION `$04`, a 16-bit accumulator read of `$C08B + n*16` pops two bytes per instruction. Read RXLO then RXHI for the count, clamp to what the frame still owes, pop count AND `$FFFE` bytes as words, and take an odd trailing byte through DATA.

```
; entry: 16-bit A, 8-bit X/Y, X = slot*16, ptr -> buffer,
;        cnt = even number of bytes already known to be queued
wide:   ldy     #0
@w:     lda     DATAW,x         ; $C08B,x: pops +B then +C
        sta     (ptr),y
        iny
        iny
        dec     cnt             ; cnt in words
        bne     @w
        rts
```

Never pop a single byte through +B or +C; the re-issue guard would treat the second single as a repeat. The wide drain measures about 80 KB/s on a stock IIGS.

In `cogslib` the slot transport's `rdn` hook uses the burst path when the caller already knows the count is queued; `cogs_recv_frame` and the HTTP collectors use it.

9.4 HTTPS in three commands

Fetching a page over TLS takes WIFI_JOIN (or stored credentials at power-on), a nonzero TIME, and HTTP. TIME is checked first because a verified TLS connect is refused until the clock is synchronized.

```
cogs_u32 t = 0;
while (cogs_time(&card, &t) == COGS_OK && t == 0) { /* wait for NTP */ }

static cogs_u8 body[4096];
cogs_http_reply r;
if (cogs_http_get(&card, 0, "https://a2cogs.com/disks/cogs-hub.json",
                  body, sizeof body, &r) == COGS_OK && r.err == COGS_E_OK) {
    /* r.status is the HTTP status, r.body_len bytes are in body,
       r.truncated is set if r.total exceeded the buffer */
}
```

`cogs_http_get` sends the HTTP command, then collects DATA events for socket 0 until HTTP_DONE or ERROR arrives, copying into the caller's buffer. For a request with headers or a body, `cogs_http_request` takes the method, extra header lines, and body. `cogs_http_begin` and `cogs_http_finish` split the two halves so the card fetches while the program does other work; no other frame-receiving call may be made between them.

The same request in raw frames is one HTTP command (Chapter 4) followed by a read loop that demultiplexes `$91`, `$95`, and `$93` on socket 0.

9.5 Getting one value out of JSON

Set RETAIN on the request so the card keeps the body, then ask for a path. The Apple II never parses JSON.

```
cogs_http_reply r;
cogs_http_request(&card, 0, "GET", "https://a2cogs.com/disks/cogs-hub.json",
                  NULL, NULL, 0, COGS_HTTP_RETAIN, NULL, 0, &r);

static cogs_u8 val[64];
cogs_json_reply j;
if (cogs_json_get(&card, 0, "cats[0].name", val, sizeof val, &j) == COGS_OK
    && j.err == COGS_E_OK) {
    /* j.copied bytes of UTF-8 in val */
}
```

`j.err` is 13, NO_RETAINED, if no retained document exists for that socket, and 14, JSON_NOPATH, if the path does not resolve. The retained document is limited to 32 KB and stays until the next retaining request.

9.6 Hashing and random numbers

Both are synchronous and need no network.

```
/* SHA-256 of a buffer */
static cogs_u8 msg[3 + 256];
msg[0] = 0;                        /* algo 0 = single SHA-256 */
msg[1] = (cogs_u8)(n & 0xFF);      /* data_len lo */
msg[2] = (cogs_u8)(n >> 8);        /* data_len hi */
memcpy(&msg[3], data, n);
```

```
cogs_send_frame(&card, 0x30, msg, 3 + n);
cogs_recv_frame(&card, &op, digest, 32, &len);    /* op 0xB0, len 32 */

/* 16 random bytes */
msg[0] = 16;
cogs_send_frame(&card, 0x31, msg, 1);
cogs_recv_frame(&card, &op, rnd, 16, &len);       /* op 0xB1, len 16 */
```

The digest is in standard (big-endian) byte order. For a Bitcoin-style double hash use algo 1.

9.7 Mounting a disk image from a program

A program can mount a volume on a handle the operating system is not using, read blocks directly, and close it, without touching the SmartPort mount table.

```
cogs_vol_reply v;
cogs_vol_open(&card, 7, 0, "http://192.168.1.20:8080/Games/Arkanoid.2mg", &v);
if (v.result == COGS_E_OK) {
    static cogs_u8 block[512];
    cogs_u8 result;
    cogs_vol_read(&card, 7, 2, block, &result);   /* volume directory */
    cogs_vol_close(&card, 7, &result);
}
```

To make a volume appear as a drive, write the mount table instead:

```
cogs_u8 result;
cogs_vol_mount_set(&card, 3, "https://a2cogs.com/disks/x.2mg", &result);
```

The change applies at once. The card raises the unit's disk-switched flag and GS/OS remounts drive 3 on the running desktop. An empty URL clears the unit.

9.8 An 8-bit host in cc65

The same `cogslib` sources compile with cc65 for ProDOS 8. The slot transport uses 16-bit absolute addresses and the 6502 polling idioms; the API is identical.

```
#include "cogs.h"
#include "cogs_io_slot.h"

static cogs      card;
static cogs_io   io;
static cogs_slot slot;

int main(void)
{
    cogs_ping_reply p;
    cogs_status_reply s;

    if (cogs_io_slot_find(&io, &slot) == 0) {
        cputs("No CoGS card.\r\n");
        return 1;
    }
    cogs_init(&card, &io);
    if (cogs_ping(&card, &p) == COGS_OK && cogs_get_status(&card, &s) == COGS_OK) {
        cprintf("CoGS %u.%02u  link %u  %u.%u.%u.%u\r\n",
                p.fw_major, p.fw_minor, s.link_state,
                s.ip[0], s.ip[1], s.ip[2], s.ip[3]);
```

```
    }
    return 0;
}
```

The CoGS Console (`COGS.SYSTEM`) on the 8-bit disk is this program grown into a menu: Wi-Fi scan and join, network detail, a connectivity test that reports the TLS session the card terminated, and a ProDOS clock set from the card's NTP time.

9.9 The `cogslib` library

`cogslib` is portable C with a transport behind two function pointers, so the same source builds for ORCA/C, cc65, and a host compiler against a software model of the card. A `cogs` session is a transport plus a poll cap; every helper returns `COGS_OK` (0) when a complete round trip happened, or a negative library code.

Table 9-1. `cogslib` return codes

Return	Meaning
`COGS_OK`	A frame round trip completed. Inspect the reply's own result field.
`COGS_ERR_TIMEOUT`	A STATUS poll spun past `spin_cap`.
`COGS_ERR_NO_CARD`	The presence probe failed.
`COGS_ERR_OVERSIZE`	A frame did not fit the caller's buffer.
`COGS_ERR_PROTO`	Unexpected opcode or malformed reply.
`COGS_NO_EVENT`	`cogs_poll_event` only: nothing was waiting.

Table 9-2. `cogslib` functions

Group	Functions
Transport	`cogs_io_slot_init`, `cogs_io_slot_find`, `cogs_io_stub_init`
Session	`cogs_init`, `cogs_reset`, `cogs_present`
Framing	`cogs_send_frame`, `cogs_recv_frame`, `cogs_poll_event`
System	`cogs_ping`, `cogs_get_status`, `cogs_netinfo`, `cogs_tls_info`, `cogs_card_info`, `cogs_time`, `cogs_prefs_get`, `cogs_prefs_set`
Wi-Fi	`cogs_wifi_scan`, `cogs_wifi_join`, `cogs_wifi_save`
Network tools	`cogs_resolve`, `cogs_probe`, `cogs_icmp_ping`, `cogs_traceroute`
HTTP and JSON	`cogs_http_get`, `cogs_http_request`, `cogs_http_begin`, `cogs_http_finish`, `cogs_json_get`
Link layer	`cogs_link_open`, `cogs_link_frame`, `cogs_link_close`, `cogs_link_ip`
Disk	`cogs_vol_open`, `cogs_vol_read`, `cogs_vol_close`, `cogs_vol_url_set`, `cogs_vol_url_get`, `cogs_vol_mount_set`, `cogs_vol_mount_get`
Catalogs	`cogs_cat_url_set`, `cogs_cat_url_get`, `cogs_aud_url_set`, `cogs_aud_url_get`
Streams	`cogs_stream_open`, `cogs_stream_read`, `cogs_stream_close`, `cogs_stream_stat`

Card error codes are the `COGS_E_*` constants (Appendix A). HTTP flags are `COGS_HTTP_STREAM`, `COGS_HTTP_HDRS`, `COGS_HTTP_RETAIN`, and `COGS_HTTP_KEEPALIVE`; CONNECT flags are `COGS_CONNECT_TLS` and `COGS_CONNECT_NOVERIFY`. The library discards frames it is not waiting for; a program that must observe events (DATA on a raw socket, WIFI_STATE) reads them with `cogs_poll_event` between commands.

CHAPTER 10

Hardware

The CoGS card is a four-layer board carrying a socketed Raspberry Pi Pico 2 W module, three bus transceivers, a W5500 Ethernet controller with magnetics and an RJ45 jack, a 3.3 V regulator, a status LED, and the Apple II slot edge connector. This chapter gives the block diagram, the pin assignment between the RP2350 and the bus, the Ethernet section, power, mechanical notes, and the differences between board revisions.

10.1 Block diagram

```
          Apple II slot edge connector (50 pins)
                          |
            74LVC245 x3  (address, data, control)
                          |
          Raspberry Pi Pico 2 W module (socketed)
            RP2350A  2 x Cortex-M33 200 MHz
            520 KB SRAM, 4 MB QSPI flash
            CYW43439  802.11n 2.4 GHz, on-module antenna
            SHA-256 peripheral, TRNG
                |                     |
             SPI0                   USB
                |                     |
          W5500 MAC/PHY         micro-USB (UF2 flash,
          25 MHz crystal         serial console)
                |
        HR911105A RJ45 magjack
        LINK / ACT LEDs
```

The RP2350′s SHA-256 engine and true random number generator, which back HASH, RANDOM, and NONCE_SCAN, and the radio are all on the module. The board adds the bus interface, the wired Ethernet path, and power.

10.2 Bus interface

Three 74LVC245 octal transceivers connect the 5 V slot bus to the 3.3 V RP2350. Twelve address lines and R/W share one group of GPIOs with the eight data lines; the transceivers' output enables select which group is on the GPIOs during each phase of the cycle, under control of PIO state machines. The three select lines /DEVSEL, /IOSEL, and /IOSTRB are combined into one ENBL signal that starts the capture.

Table 10-1. RP2350 to bus pin assignment

Signal	GPIO	Direction	Function
ENBL	2	in	/DEVSEL AND /IOSEL AND /IOSTRB; falling edge starts a cycle
A0 to A11, R/W	3 to 15	in	Address and R/W, latched at ENBL fall (R/W on 15)
D0 to D7	3 to 10	in/out	Data, shared with the address group
PHI1	16	in	Bus clock phase, used to time the data drive
/RESET	17	in	Slot reset, taken as an interrupt
IRQ	18	out	Slot interrupt request, active high at the GPIO; not asserted by the current firmware
ADDR_OE	26	out	Address transceiver output enable
DATA_OE	27	out	Data transceiver output enable
DATA_DIR	28	out	Data transceiver direction

On a slot read the address is latched on the falling edge of ENBL, decoded by the state machine, and the staged byte for that register or ROM address is driven onto the data transceiver before the processor latches the bus. On a write the data byte is captured at the end of the cycle and queued to core 0. Core 1 services this path alone and never blocks.

The card decodes three regions: the sixteen registers on /DEVSEL, the 256-byte `$Cn00` page on /IOSEL, and the 2 KB `$C800` window on /IOSTRB. Which slot the card is in is not wired to the card; the slot hardware decodes it, and the slot ROM discovers its own slot number from the return address of a JSR into its `$Cn00` page.

10.3 Ethernet

The wired port is a WIZnet W5500 on SPI0, operated in MACRAW mode as a plain MAC and PHY. Its hardwired TCP/IP is not used; the card's single lwIP stack sees it as a second network interface beside the radio, so TLS, HTTP, volumes, and card services do not know which medium carries their traffic. The chip is polled every 200 ms; three dead reads declare it lost and re-initialise it. A rail dip on a GS reset can reset the W5500 between two polls, which clears its MAC register (SHAR) and socket buffer split while the hardware MAC filter stays on, so every unicast reply to the card is filtered and DHCP retries forever. From firmware 1.26 the poll reads SHAR back and, if it is not the card's MAC, re-programs SHAR, the 16 KB + 16 KB socket 0 buffers and MACRAW in one go (`eth chip reset behind us`); the lease and lwIP state are untouched.

Table 10-2. W5500 SPI connections

Signal	GPIO
SCK	22
MOSI	19
MISO	20
/CS	0
/INT	1
/RST	RC power-on reset, not a GPIO

The analog section follows the WIZnet reference design for a W5500 with an integrated-magnetics jack: 49.9 Ω terminations on each side of the TX and RX pairs, a 10 Ω feed to the transmit center tap, 6.8 nF coupling on the receive pair, and a 1 nF 2 kV capacitor from the jack shield to chassis ground. The W5500 has a 25 MHz crystal and a 12.4 kΩ 1 % RSET resistor. The jack's LINK (green) and ACT (yellow) LEDs are driven by the W5500 through 330 Ω resistors.

The SPI link runs at 24 MHz. The RP2350 peripheral clock is 48 MHz at boot and the SPI block divides it by an even number, so the available rates are 24, 12, and 8 MHz; 24 MHz measured 1.4 to 1.9 MB/s of raw TCP payload through the card on three boards with no SPI faults, and a 2x margin against the rate at which the chip first misbehaves. The PHY autonegotiates. Forcing 100 Mbit full duplex against an autonegotiating switch makes the switch fall back to half duplex, which loses about a tenth of inbound packets to collisions; the card does not do that.

The wired port is preferred when its cable is up and it has a DHCP lease. Wi-Fi stays associated and takes the default route back if the cable drops. NETINFO reports the medium that is routing; an SSID length of 0 with link state 2 means wired.

10.4 Power

The slot supplies 5 V. A TPS563203 buck regulator produces the 3.3 V rail (+3V3E) for the W5500, its PHY, and the jack LEDs; a ferrite bead separates the analog PHY supply (+3V3A) from it. The Pico module takes 5 V on VSYS through a power-select header (J1) and regulates its own 3.3 V for the RP2350, the flash, and the radio. With J1 open the module can run from USB alone on the bench, with the bus interface unpowered.

The card draws its full current from the slot's 5 V supply. Wi-Fi transmit bursts and the W5500 dominate; the RP2350 at 200 MHz is a small part of the total.

10.5 Mechanical

The board is a standard-height Apple II peripheral card, four layers, 1.6 mm, with a beveled 50-pin edge connector. The Pico module sits in two 20-pin sockets (SKT1, SKT2) and is removable. The RJ45 jack is a low board-mount part on the front edge of the card, where a IIGS case leaves room for a patch cable; it does not face the rear. All surface-mount parts are on the top side. The micro-USB connector and BOOTSEL button are the module's own, reachable with the card in a slot.

10.6 Board revisions

Every revision reproduces the same bus interface, pin assignment, and register behavior, and the firmware cannot distinguish them from one another or from a Pico 2 W on an A2Pico development carrier.

Table 10-3. Board revisions

Revision	Ethernet	Notes
1.0	Discrete W5500, RJ45, microSD	First assembled board.
2	WIZ850io module on headers	Module height and rear-facing jack; microSD retained.
3	Discrete W5500, HR911105A jack	Jack moved to the front edge; microSD removed.
3.1	As 3, magnetics values revised	
3.2	Ethernet analog section redrawn from the WIZnet reference	All SMD on the top side. Production.

The A2Pico carrier (Pico 2 W on the classic 74LVC245 A2Pico board) runs the same firmware with no wired port. Firmware built for the A2Pico 2 Lite family uses a different GPIO map and is a separate UF2; the Firmware Update page matches the manifest's board tag to the running board so one is never flashed with the other.

CHAPTER 11

The Emulator

A software model of the card exists as a MAME slot device, so software for the card can be developed and tested in an emulated Apple II with no card in a slot. The same device is built into a CoGS-enabled copy of the Ample front end on macOS. This chapter describes what the model implements, how it differs from the card, how to install it, and how to script it for automated tests.

11.1 What the model implements

The device is an `a2bus` card named `cogs`, listed as **CoGS Co-Processor & Network Card** in MAME's slot device list and in Ample's slot pop-up, and accepted in any slot 1 to 7. It decodes the sixteen registers at `$C0n0` to `$C0nF`, the `$Cn00` page, and the `$C800` window, and moves bytes through the same two FIFOs the card has.

```
65816  ->  $C0nX registers  ->  cogs.cpp (a2bus device: registers, FIFOs, ROM views)
                                     |
                                 frames.c   (frame parser)
                                     |
                                 commands.c (dispatcher)  --  json.c, config_ui.c, ...
                                     |
                                 cogs_backend.c (host network engine, settings, cache)
                                     |
                                 reply frames  ->  RX FIFO  ->  $C0nX DATA reads
```

The register shell is the only emulator-specific code. The frame parser, the command dispatcher, the JSON walker, the config menus, the stream engine, the equalizer, and the catalog parsers are the firmware's own C source files, compiled into the device unchanged. The wire behavior of the model and the card cannot diverge because it is one implementation. The slot ROM is the same assembled image the card serves.

Behind the dispatcher a host backend replaces the firmware's radio and lwIP: POSIX sockets for TCP and DNS, OpenSSL for TLS, the host clock for TIME and NTP, a software SHA-256 for HASH, the host random source for RANDOM. Asynchronous operations resolve on a timer the device runs, so CONNECTED, DATA, HTTP_DONE, and the deferred disk and stream replies arrive with the same event shapes as on the card.

Table 11-1. Model coverage

Feature	Model
Registers and FIFOs	Complete, including the wide window, BULK, the CDA register, VOLIO, SYNC, the unit status view, and all `$C800` views
Slot ROM, boot, SmartPort	The card's ROM image
Wi-Fi	A fixed scan list; WIFI_JOIN succeeds without a radio
TCP, DNS, HTTP, TLS	Real host sockets; `https://` and TLS sockets work; certificates are not verified
Disk volumes	Complete, with the read-ahead window and a flash cache emulated in a host file
Byte streams, MP3 decode, EQ	Complete
Card services	HASH, RANDOM, JSON_GET complete; NONCE_SCAN returns UNSUPPORTED
Config terminal, desk accessory install, tool sets	Complete
Link layer	Bridge mode over the host's virtual network interface, in the Ample build
Firmware update	The Check step only; there is no image to flash

Settings the card keeps in flash (credentials, mount table, catalog bases, prefs blob, card settings) are kept in a host file, `~/.cogs_emu_nvram` by default, so a save followed by a restart of the emulator reproduces the card's power-cycle behavior. The recent-disk list is `~/.cogs_emu_recent.json`, with `name`, `url`, `boot`, `tr`, and `ys` on each item. The flash warm cache is a host file, `~/.cogs_emu_flashcache`.

11.2 Differences from the card

The model is faithful to the protocol and the software, not to the bus.

- **No bus timing.** Register reads and writes are instantaneous. There is no 500 ns deadline and no way to observe it.
- **No dummy cycles.** MAME's 65816 does not present the indexed-store dead cycle or the 16-bit-index cycle 3a to device handlers. Host code that violates the rules in Chapter 8 runs correctly in the emulator and fails on a stock machine. The emulator cannot be used to prove those rules are followed.
- **No accelerator re-issue.** Stretched I/O cycles do not occur.
- **Network is the host's.** Latency, bandwidth, and DNS are those of the Mac. A `COGS_VOL_LATENCY_MS` setting adds a fixed delay per block fetch for testing.
- **TLS is unverified.** The backend does not check certificate chains; CERT_VERIFY is never returned. The card's mbedTLS path with the bundled roots is the verified one.
- **No radio.** RSSI is a constant, WIFI_SCAN returns a fixed list, and joins cannot fail unless `COGS_EMU_JOIN_FAIL` is set.
- **No mining.** NONCE_SCAN returns UNSUPPORTED; the card's SHA-256 peripheral has no host equivalent worth modeling.
- **Speed.** MAME runs the 65816 at 2.8 MHz by default. `COGS_CPU_SCALE` raises it (2.5 approximates an accelerator), which changes which audio paths the desk accessory selects.

11.3 Installing in Ample

Ample launches a MAME binary inside its application bundle and builds each machine's slot pop-up from a property list in the bundle. Installing the model is three steps, performed by `emu/mame/ample_push.sh` after the device has been built with `build_ample_mame.sh` against Ample's own MAME fork:

1. Copy the built `mame64` over `Ample-CoGS.app/Contents/MacOS/mame64` and code-sign it with `mame64.entitlements`, which allows the binary to load the bundle's SDL framework.
2. Run `ample_patch_plist.py` on every `apple2*.plist` in the bundle so the card appears in the slot pop-up for every slot with the current version in its name.
3. Re-sign the application bundle.

Ample must not be running while its bundle is modified. After the push, choose **apple2gs**, set a slot's card to **CoGS Co-Processor & Network Card**, and set that slot to **Your Card** in the emulated Control Panel as on a real machine. Boot from the card by setting the startup slot to the card's slot.

The CoGS-enabled bundle is `~/Applications/Ample-CoGS.app`; the stock Ample is left untouched.

11.4 Running headless

The MAME binary inside the bundle runs from the command line with no window, which is how automated tests and the manual's screenshots are produced:

```
mame64 apple2gs -sl5 cogs -nvram_directory nv -video none -sound none \
    -skip_gameinfo -snapshot_directory shots \
    -autoboot_script test.lua -seconds_to_run 300
```

`-sl5 cogs` places the card in slot 5. `-nvram_directory` selects the emulated machine's battery RAM, which must already hold **Your Card** for that slot; the test scripts seed it. `-autoboot_script` runs a Lua script against the machine.

Table 11-2. Emulator environment variables

Variable	Effect
`COGS_EMU_NVRAM`	Path of the card settings file
`COGS_EMU_RECENT`	Path of the recent-disk list
`COGS_EMU_FLASHCACHE`	Path of the flash cache file
`COGS_VOL_URL`	Unit 1 image URL, overriding the stored one
`COGS_CAT_URL`, `COGS_AUD_URL`, `COGS_PIC_URL`	Catalog base URLs, overriding the stored ones
`COGS_EMU_SHARES`	`Name=url;Name2=url2`, the LAN shares the Browse pages list
`COGS_FORCE_LINK`	Bring the link up at power-on without a join
`COGS_EMU_JOIN_FAIL`, `COGS_EMU_JOIN_SLOW`	Make WIFI_JOIN fail or take time
`COGS_EMU_SWITCHED`	Start with unit 1′s disk-switched flag raised
`COGS_VOL_LATENCY_MS`	Fixed delay per block fetch
`COGS_CPU_SCALE`	Processor speed multiplier
`COGS_FIFO_TRACE`	One log line per request parsed and reply pushed
`COGS_VOL_TRACE`	Log every volume operation
`COGS_CFG_TRACE`	Log every CONFIG key byte

The device also appends a line to `/tmp/cogs_cda_trace.log` for each desk accessory install milestone, which is the quickest confirmation that the card is in the slot and the installer ran.

11.5 Scripting with Lua

MAME's Lua interface exposes the emulated machine's memory, keyboard, and video. A test script polls the text screen for expected strings, posts keystrokes, and takes snapshots. The pattern used by the scripts under `emu/mame/test/` is:

```lua
local sp = manager.machine.devices[":maincpu"].spaces["program"]

-- read the 40-column text page as 24 strings
local function taddr(r, c)
    return 0x400 + (r % 8) * 0x80 + math.floor(r / 8) * 0x28 + c
end
local function screen_has(needle)
    local rows = {}
    for r = 0, 23 do
        local t = {}
        for c = 0, 39 do
            local b = sp:read_u8(taddr(r, c)) & 0x7f
            if b < 0x20 then b = b + 0x40 end
            t[c + 1] = string.char(b)
        end
        rows[r + 1] = table.concat(t)
    end
    return table.concat(rows, "\n"):upper():find(needle:upper(), 1, true) ~= nil
end

-- type one character through the keyboard latch
local function post(ch)
    manager.machine.natkeyboard:post(ch)
end

-- one step per frame
emu.register_frame_done(function()
    if state == "menu" and screen_has("CoGS Config") then
        manager.machine.video:snapshot()
        post("2")
        state = "browse"
    end
end)
```

`manager.machine.video:snapshot()` writes a PNG into `-snapshot_directory`. Scripts end the run with `manager.machine:exit()` or by letting `-seconds_to_run` expire, and report pass or fail on standard output for the shell harness that launched them. Card-side state (the settings file, the trace log, the share server's request log) is inspected by the harness after MAME exits.

11.6 Building

`emu/mame/integrate.sh` fetches upstream MAME, copies the device sources and the shared firmware files in, applies the two small patches that register the device (`cards.cpp` for the slot device list and `scripts/src/bus.lua` for the build), and builds an Apple II-only binary suitable for continuous integration. `build_ample_mame.sh` does the same against Ample's fork with its menus and virtual network support, producing the binary `ample_push.sh` installs. `test/devtest.c` compiles the device logic with the host backend into a standalone program that exercises the protocol, including loopback sockets and HTTP, without MAME.

Appendix A. Error Codes

Card error codes

The first byte of every reply frame is a code from this table. Zero is success; the reply's remaining payload is then as documented for the command. Nonzero codes carry no further payload unless the command's description says otherwise.

Table A-1. Card error codes

Code	Name	Meaning
0	OK	Success.
1	NOT_JOINED	No network link. WIFI_JOIN has not completed, or the link dropped.
2	DNS_FAIL	The host name did not resolve.
3	CONN_TIMEOUT	The TCP connection or TLS handshake did not complete in time.
4	CONN_REFUSED	The peer refused the connection, or reset it.
5	TLS_HANDSHAKE	The TLS handshake failed for a reason other than certificate verification.
6	CERT_VERIFY	The server's certificate did not verify against the card's root store, or the name did not match.
7	NO_FREE_SOCK	All eight socket handles are in use.
8	SOCKET_STATE	The handle is not in a state that permits the operation.
9	BUF_OVERRUN	The payload exceeds the command's limit, or a send would exceed the card's buffer.
10	BAD_FRAME	The frame length or payload is malformed for the opcode.
11	UNSUPPORTED	Unknown opcode, or the operation is not available: a write to a read-only volume, NONCE_SCAN in the emulator, a stream kind the card cannot decode.
12	INTERNAL	A firmware error not covered above, including a firmware update whose image hash did not match.
13	NO_RETAINED	JSON_GET with no retained HTTP response.
14	JSON_NOPATH	JSON_GET path did not resolve in the retained document.
15	BUSY	The service is in use: a nonce scan is running, a stream is already open on the handle, a volume operation is in flight.
16	VOL_NOMOUNT	Volume operation on a handle with no open volume.
17	VOL_RANGE	Block number beyond the volume's end.
18	STREAM_NONE	Stream operation on a handle with no open stream.
19	WIN_SHORT	VOL_WRITE through the block window, but fewer than 512 host stores landed since the window was armed. The host re-sends the block through the FIFO.
20	VOL_DUP	VOL_MOUNT_SET refused because that image is already mounted on another unit.

Codes are stable across firmware versions. New codes are appended; existing values are not reassigned.

SmartPort and ProDOS error mapping

The slot ROM translates card results into the error codes the operating system expects. The ROM returns these in the accumulator with the carry set.

Table A-2. Error mapping in the slot ROM

Condition	ProDOS / SmartPort code	Name
Card reply OK	`$00`	No error
Unit number greater than the unit count	`$28`	No device connected
Unit present but no image mounted	`$27`, STATUS reports offline	I/O error, no media
Read failed after the ROM's retries	`$27`	I/O error
Write to a read-only mount (card reply UNSUPPORTED)	`$2B`	Write protected
Block past the end of the volume (card reply VOL_RANGE)	`$27`	I/O error
SmartPort command number not 0 to 4	`$01`	Bad command
SmartPort STATUS or CONTROL code not implemented	`$21`	Bad status or control code
Disk switched since the last acknowledgment	Reported in STATUS, then cleared	

Read-only status is also advertised ahead of any write. SmartPort STATUS returns the write-protect bit for a read-only mount, and the ProDOS STATUS call returns `$2B`, so GS/OS marks the volume locked when it is mounted and does not attempt writes.

HTTP status

HTTP_DONE reports the server's status code in its payload; the card does not translate HTTP status into an error code. A request that completes with a 4xx or 5xx status still ends in HTTP_DONE, with the status in the event. Transport failures (DNS, connect, TLS) end in ERROR with a code from the card table above and no HTTP_DONE.

Volume fetch failures

A block fetch that fails is retried by the card, and the ROM retries the request in turn. When the retries are exhausted the ROM returns `$27` and the card marks the unit offline; the next request re-opens the origin.

Appendix B. URLs, File Types, and Catalog Formats

URL schemes the card accepts

Table B-1. URL schemes

Scheme	Use
`http://`, `https://`	Disk images, audio and picture files, catalog pages, HTTP requests
`tool://NNN?base=XXXXXX`	Card-served tool set NNN relocated for address XXXXXX, opened on stream handle 1 (Chapter 7)

URLs are limited to 255 bytes. Characters outside the unreserved set must be percent-encoded by the client; the card sends the path as given. Host names resolve through the DNS server from the DHCP lease. A URL with no explicit port uses 80 or 443 by scheme.

Disk image formats

Table B-2. Disk image formats

Extension	Format	Handling
`.2mg`	2IMG (Universal Disk Image)	Detected by the `2IMG` magic in the first 64 bytes, not the extension. The data offset is read from header offset `$18` (minimum 64) and the block count from `$14`. ProDOS order only.
`.po`, `.hdv`	Raw ProDOS-order blocks	The block count is the file size divided by 512.
`.dsk`, `.do`, `.woz`, `.nib`	DOS-order and nibble formats	Not block devices; not served by the Image Manager and not mountable.

The file size comes from the `Content-Range` total of the probe response. The 2IMG block count is treated as advisory and clamped to the blocks the file physically holds after the data offset. Images up to 65,535 blocks (32 MB) are supported. A volume is writable when its origin accepts PUT; the first refused write latches the mount read-only, and that state is advertised through SmartPort STATUS and the ProDOS status call. Writes never touch a 2IMG header.

The origin must support HTTP range requests and answer a ranged GET with 206. The probe requests the first 64 bytes; a 200 response is accepted for the probe (the connection is closed after the header) but a block read answered with 200 fails, since the response would be the whole file. Redirects are followed and the final URL is cached for the life of the mount so later block fetches skip the redirect chain.

Audio and picture file formats

Table B-3. Audio and picture formats

Kind	Format	Notes
PCM track	Unsigned 8-bit, 21,973 Hz, mono or planar stereo	Sample rate is fixed by the playback tool set; `.pcm` files are raw. Catalog `ch` 2 marks stereo, `hr` 1 marks the dual-oscillator hi-res encoding.
SoundSmith	SSP1 pack, catalog `kind ss`	Song and instruments in one file; played by tool set 226.
MP3 stream	MPEG-1 Layer III	Decoded on the card (minimp3) and resampled to the PCM format above, one stream at a time; used for Internet radio.
SHR picture	32,768 bytes: 32,000 pixel bytes, 200 SCBs, padding, 16 palettes; `.shr`, `.pic`, `.sh`	The card displays the first 32,712 bytes (pixels, SCBs, palettes).
HGR picture	8,192 bytes, one hi-res page; `.hgr`	Unpacked. 7,680-byte packed pages are not accepted.
DHGR picture	16,384 bytes, auxiliary half first; `.dhgr`, `.dhr`	Unpacked.

Picture kind is taken from the extension when it is one of those listed, otherwise from the size (32 KB, 16 KB, 8 KB). `.pnt`, `.sh2` (packed SHR), and `.3200` are recognized in listings but not displayed.

Catalog pages

A disk source is a base URL ending in `/`. The card appends fixed file names to it, so any HTTP server that serves the files below is a valid source.

Table B-4. Catalog requests

Request	Contents
`{base}cogs-volumes.json`	Disk catalog, page 1
`{base}cogs-volumes-N.json`	Disk catalog, page N
`{base}cogs-hub.json`	Category hub for the a2cogs catalog (fixed categories)
`{base}cogs-your-hub.json`	Category hub for a share (folder categories, firmware 1.13)
`{base}cogs-audio.json`, `-N.json`	Audio catalog
`{base}cogs-stations.json`	Internet radio stations
`{base}cogs-pictures.json`, `-N.json`	Picture catalog
`{base}cogs-botw-weeks.json`	Picture-of-the-week index (a2cogs pictures source)
`{base}search?q=…&p=N[&i=a|s|p][&cat=…][&host=8|gs]`	Search, one page of results

Page shape

Every catalog page and search result is one JSON object with a header and one array of entries. The card's parser is a streaming key scanner, not a general JSON parser: it recognizes the keys below by exact name, ignores unknown keys, and commits an entry when it reaches that entry's `url`. Within an entry, `name` must precede `url`. Whitespace and key order are otherwise free.

```
{ "pages": 122, "total": 2192, "next": "cogs-volumes-3.json",
  "catalog": "CoGS streamable disks (page 2 of 122)",
  "volumes": [
    { "name": "Airheart", "bytes": 33553920, "blocks": 65535,
      "readonly": true, "boot": true, "cat": "G", "host": "any",
      "in": "Total Replay v6.1", "source": "github/a2-4am",
      "cover": "https://a2cogs.com/pictures/tr/Airheart.shr",
      "url": "https://…/Total.Replay.v6.1.hdv" }
  ] }
```

Table B-5. Catalog keys

Key	Where	Meaning
`pages`	header	Number of pages; the card pages with N or `-N.json`.
`total`	header	Entry count across all pages. Shown on the All row.
`count`	header	Result count, on a search page.
`next`	header	Next page's file name. Informational; the card computes page names itself.
`volumes`	header	Entry array for disks. `tracks` is the entry array for audio and pictures.
`name`	entry	Display name, up to 30 characters shown.
`url`	entry	Absolute URL of the file. Commits the entry.
`bytes`	entry	File size. Audio and pictures show it; disks show `blocks`.
`blocks`	entry	Disk block count.
`boot`	entry	`true` when block 0 boots; drawn as a marker on the row.
`readonly`	entry	`true` when the origin refuses writes.
`cover`	entry	Absolute URL of a box-art SHR for the V key.
`cat`	entry	Disk category letter, one character.
`host`	entry	`gs` when a IIGS is required, `any` otherwise. Filters the 8-bit view.
`in`	entry	Name of the bundle image this title is part of. Nonempty paints TR. The `url` is the parent image.
`source`	entry	Provenance label, short.
`secs`	audio entry	Duration in seconds.
`rate`	audio entry	Sample rate; informational, playback is fixed at 21,973 Hz.
`ch`	audio entry	1 mono, 2 planar stereo.
`hr`	audio entry	1 for the hi-res dual-oscillator encoding.
`kind`	audio entry	`pcm` (default) or `ss` for SoundSmith.
`radio`	station entry	Marks a station in `cogs-stations.json`.

Pages hold 18 disk entries or 17 audio or picture entries. Larger pages are accepted; the card lists the first entries that fit its buffer and pages by the file name sequence.

Category hubs

`cogs-hub.json` describes the fixed categories of the a2cogs catalog by letter. Each row commits on `n8`.

```
{ "total": 2192, "total8": 890, "totalgs": 1310, "cats": [
  { "k": "G", "name": "Play a game", "n": 1177, "n8": 527 },
  { "k": "S", "name": "Utilities & work", "n": 202, "n8": 9 } ] }
```

`n8` and `total8` are the counts of entries with `host any`, shown in place of `n` and `total` on an 8-bit host. A category row lists through `search?q=&p=1&cat=G` (with `&host=8` on an 8-bit host).

`cogs-your-hub.json` describes a share's own categories (firmware 1.13). The card requests it when Y is pressed on a Browse hub whose source is the share, once per session per source and kind:

```
GET {base}cogs-your-hub.json          disks
GET {base}cogs-your-hub.json?i=a      audio
GET {base}cogs-your-hub.json?i=p      pictures
```

```
{ "hub": "your", "kind": "disks", "total": 41, "cats": [
  { "name": "Games",      "cat": "Games",                "n": 12 },
  { "name": "Project 17", "cat": "Utilities/Project 17", "n": 4 } ] }
```

`name` is drawn as sent, up to 30 characters. `cat` is opaque to the card and is returned percent-encoded in the `cat=` search parameter. `n` is the row count; `total` is the All row. At most twelve categories are read. A 404, an empty `cats`, or a body that does not parse mean no categories, and Y opens the share's plain A to Z list. A category row lists through `search?q=&p=1&cat=Utilities%2FProject%2017` (with `&i=a` or `&i=p` for the other kinds), which returns a normal page. `q=` and `cat=` combine; Find inside a category searches that category only.

Search

`search` returns one page in the page shape, with `count` in place of `total`. Parameters:

Table B-6. Search parameters

Parameter	Meaning
`q`	Substring, matched case-insensitively against the display name and file name. Empty matches everything.
`p`	Page number, from 1.
`i`	Index: absent for disks, `a` audio, `s` stations, `p` pictures.
`cat`	Category letter (a2cogs) or opaque path (share).
`host`	`8` or `gs`, filters disks by `host`.
`week`	Picture-of-the-week number, with `i=p&cat=M`.

A server that ignores a parameter it does not know returns an unfiltered result and the card lists it.

The share protocol

The Image Manager and any compatible share serve the requests above from a folder, plus the files themselves:

Table B-7. Share requests

Request	Response
`GET /`	Index page, HTML, with links to the catalogs.
`GET`, `HEAD` on a file path	The file. `Range` is honored with 206. Paths are relative to the shared folder and percent-encoded.
`PUT` on a disk image path with `Content-Range: bytes a-b/*`	Writes the range into the image when the share is writable; 200 on success, 403 when the share is read-only, 416 for a range outside the file.
`GET /cogs-volumes.json`, `-N.json`, `/search`, `/cogs-your-hub.json`	Catalogs, as above.
`GET /cogs-audio.json`, `-N.json`, `/audio/…`	Audio catalog and PCM or SoundSmith files.
`GET /cogs-pictures.json`, `-N.json`, `/pictures/…`	Picture catalog and SHR files.
`GET /cogs-stations.json`	An empty station list (`"tracks": []`).

Entries in a share catalog carry `"source": "lan"` and `"readonly"` according to the share's writable setting.

Disk image responses (GET, HEAD, and the 206 range replies) may carry two headers. `X-CoGS-Writable: 0|1` is the share's current writable setting; the card latches or clears the unit's read-only flag from it and raises disk-switched when it changes. `X-CoGS-Poll: N` asks the card to re-fetch 64 bytes of each mounted image every N seconds (2 to 60) while the bus is idle, so the header is re-read without waiting for the Apple II to touch the disk; three consecutive unanswered polls dismount the unit (Chapter 6). Shares that omit the headers are never polled. The Image Manager sends `X-CoGS-Poll: 5`. Audio and picture entries served from a share use `/audio/` and `/pictures/` paths that include the subfolder, so files with the same name in different folders do not collide.

Shares announce themselves on the LAN with mDNS service type `_cogs-share._tcp` on the share's port (default 8271). The service instance name is the share's display name; the TXT record carries `path=/`. The card lists discovered shares on the Browse hubs' source screens.

Appendix C. Glossary

2IMG

The Universal Disk Image container: a 64-byte header followed by the disk data. The card reads the data offset and block count from the header.

Block

512 bytes, the unit of disk transfer on the Apple II.

Block window

The 512-byte view of one disk block at `$C800` to `$C9FF`, mapped by CONTROL bit 7. Reads deliver a fetched block without FIFO pops; stores fill a block for VOL_WRITE.

BULK

The read-only register at `$C0n7` that reads `$FF` when 512 or more bytes are queued for the host, so a block drain can skip per-byte STATUS polls.

Capability bitmap

The byte returned by PING that says which command families the running firmware implements. Host code tests it instead of comparing version numbers.

Catalog

A JSON listing of disk images, audio, or pictures at a source URL, paged in fixed-name files (`cogs-volumes.json`, `cogs-volumes-2.json`, and so on).

Recent-disk list

Twelve last-used volume URLs the card keeps in flash. Browse hub R and Disk Drives R. No opcode. Chapter 6.

CDA

Classic desk accessory. A small IIGS program on the text screen reached with Ctrl-OpenApple-Esc. **CoGS Config** is the card's CDA; the card injects it at boot.

Category, hub

A grouping of catalog entries shown on a Browse page. The a2cogs catalog has fixed categories by letter; a share defines its own from folders (the "your hub").

CONTROL

The write-only register at `$C0n4`: pulse bits for soft reset, FIFO flushes, and the `$C800` view selects.

Dead cycle

The extra bus read a 6502-family processor performs during an indexed store or a page-crossing indexed load, at an address the program did not name. Chapter 8.

DEVSEL, IOSEL, IOSTRB

The three slot select signals. /DEVSEL decodes the sixteen registers at `$C0n0`, /IOSEL the `$Cn00` page, /IOSTRB the shared `$C800` window.

Deferred reply

A command whose reply is not staged immediately (WIFI_JOIN, VOL_MOUNT_SET, STREAM_OPEN). The host sees STATUS bit 7 rise later; the reply frame arrives with the same opcode as an immediate one.

Event

A frame from the card that no command asked for: CONNECTED, DATA, HTTP_DONE, ERROR, and the rest of Chapter 5.

FIFO

First in, first out queue. The card has one in each direction; the host writes the TX FIFO and reads the RX FIFO through DATA and the wide window.

Frame

One protocol message: an opcode byte, a two-byte little-endian length, and the payload.

Handle

A small integer naming one of eight sockets, eight mount units, or two streams.

Image Manager

The CoGS Image Manager, a macOS application that shares a folder of disk images, audio, and pictures as a source, and defines its categories.

Injection

The card's practice of placing code in the IIGS at boot: the CDA installer through the `$C800` overlays, and tool sets 219 and 226 through streams.

Mount table

The eight unit slots, each holding an image URL, its block count, its read-only flag, and its disk-switched flag. Persisted in flash.

MACRAW

The W5500 mode in which the chip is a plain Ethernet MAC and the card's own IP stack handles everything above it.

Range request

An HTTP request for part of a file by byte offset (`Range: bytes=a-b`), answered with 206. The card fetches disk blocks and picture and audio data this way.

Read-ahead window

The run of consecutive blocks the card fetches after a block request, in anticipation of sequential reads.

Register base

`$C080 + slot × 16`, the address of DATA for a card in that slot. Held in X by the slot ROM.

Retained response

An HTTP response body kept on the card after HTTP_DONE (the RETAIN flag) for JSON_GET queries.

SCB

Scan-line control byte. One per Super Hi-Res line, choosing the palette and mode for that line.

SHR

Super Hi-Res, the IIGS graphics mode: 320 by 200 pixels, 16 colors per line from 16 palettes, in a 32 KB frame.

Slot ROM

The 256-byte `$Cn00` page and the 2 KB `$C800` window the card serves: boot code, the SmartPort and ProDOS entry points, and the menu loader.

SmartPort

Apple's block device protocol for slot cards. The card presents its drives through SmartPort and the ProDOS block entry points.

Source

A base URL the Browse pages read catalogs from: the a2cogs catalog, a share, or any HTTP server holding the catalog files.

STATUS

The read-only register at `$C0n1`: flags for RX_AVAIL, TX_READY, LINK_UP, and the sticky ERROR latch.

Stream

A byte source opened by URL on one of two stream handles, drained through the FIFO or the wide window: audio tracks, radio, pictures, and tool set images.

Tool set

A numbered library of routines in the Apple IIGS Toolbox. The card supplies tool sets 219 and 226.

UF2

The file format the RP2350 bootloader accepts over USB. Firmware releases ship as `cogs-<version>.uf2`.

Unit

One of the eight SmartPort units the card presents, numbered 1 to 8. Unit N maps to mount handle N minus 1.

Unit status view

Registers `$C0nD` to `$C0nF`: select a mount handle, read its block count and flags, acknowledge its disk-switched flag.

Wide window

The pop registers at `$C0nB` and `$C0nC` (DATAW_L, DATAW_H). Each read pops one byte from the same stream as DATA, so a 16-bit load takes two bytes in one instruction.

Your hub

The category screen a share defines from its folders, reached with Y on a Browse hub. Firmware 1.13.

Appendix D. Credits and Licenses

The CoGS project

CoGS is an open source, open hardware project designed and developed by Rob Perissi. Firmware, card ROM, GS/OS software, the Image Manager, the emulator device, and the hardware design files are published at github.com/rperissi/cogs under the project license stated there.

Hardware lineage

Table D-1. Hardware lineage

Component	Origin	License
Bus interface, PIO bus library	A2Pico, Oliver Schmidt (github.com/oliverschmidt/a2pico)	MIT
A2Pico carrier hardware	Ralle Palaveev	Open hardware
Ethernet analog section	WIZnet W5500 reference design	Vendor reference

Third-party software in the firmware

Table D-2. Firmware components

Component	Author	License
pico-sdk, with the CYW43 driver	Raspberry Pi Ltd and contributors	BSD-3-Clause
lwIP	The lwIP project, Swedish Institute of Computer Science	BSD-3-Clause
mbedTLS	Arm Ltd and contributors	Apache-2.0
minimp3	Lion (github.com/lieff/minimp3)	CC0
Root certificate bundle	Mozilla NSS, via curl's extraction	MPL-2.0

Catalog content

Disk images in the a2cogs.com catalog are mirrored from public archives and identified by source on every entry. The IIGS titles come from the What is the Apple IIGS? collection (whatisthe2gs.com); the 8-bit arcade titles are streamed from Total Replay by 4am (github.com/a2-4am/4cade). Audio tracks carry their attributions in the audio catalog. Pictures of the week are credited to their artists in the picture catalog.

Software served to the IIGS

Table D-3. IIGS software

Component	Author and lineage	Terms
Tool 226, PCM streaming player	Petar Puskarich's Tool225 (github.com/ppuskari), on NinjaTracker Plus by Ninjaforce, toolified by Brutal Deluxe, on FTA and ESP	Used with permission
Tool 219, SoundSmith player	Huibert Aalbers (player), Olivier Goguel and FTA (tool set), Brutal Deluxe Software (v2.1)	Freeware, unmodified
Marinetti link layer	CoGS project, for Marinetti by Richard Bennett and Andrew Roughan	Same as the project

Development tools

MAME and the Ample front end by Kelvin Sherlock host the emulator device. ORCA/C, Golden Gate, cc65, ca65, Merlin 32, gsasm, Typst, and Pandoc were used to build the software, the tool sets, and this book. FujiNet's design and documentation informed the card services.

Trademarks

Apple, Apple II, Apple IIGS, GS/OS, ProDOS, and Finder are trademarks of Apple Inc. Raspberry Pi is a trademark of Raspberry Pi Ltd. Other names are the property of their owners.