TECHNICAL SPECIFICATION

UDP DAC Streaming Protocol Specification

Protocol v1.0 · WiFiCube / WiFiCube Ultra MK2

PROTOCOL VERSION	v1.0
DOCUMENT VERSION	v1.0
DATE	29 April 2026
AUTHOR	Paul Mainwaring
APPLIES TO	WiFiCube and WiFiCube Ultra MK2
AUDIENCE	Software engineers integrating a host application that streams DAC s data to a WiFiCube and WiFiCube Ultra MK2 over UDP
STATUS	Implementer reference — normative

Table of Contents

1. Overview

Both the WiFiCube and the WiFiCube Ultra MK2 implement a three-port UDP protocol for host control and DAC streaming. The protocol on the wire is identical across both products and has been stable since the first firmware release. Each port serves a distinct purpose and may be used independently:

UDP Port	Channel	Purpose
45456	Discovery / liveness	Host pings the device to detect presence and read its unique ID. Stateless request/reply.
45457	Control (commands)	Configuration, status queries, security operations. Each request elicits at most one response.
45458	DAC sample streaming	Continuous unidirectional flow of XY/RGB samples from host to device. Optional per-packet status reply for flow control.

All three ports listen on every interface (`INADDR_ANY`, IPv4) once the device has finished bringing up the network. The device may be reached via its WiFi AP, WiFi station, or wired Ethernet (W5500) interface — the wire format is identical on every transport.

> *All multibyte integers in this specification are **little-endian** unless explicitly stated otherwise.*

2. Transport and Endpoints

The protocol uses UDP/IPv4 datagrams. The device sets the `IP_TOS` traffic class on the streaming socket to **AC_VI (Video, DSCP class 4)** to encourage favourable queuing on Wi-Fi APs that honour WMM. Hosts **should** mark outgoing streaming packets the same way; control and discovery packets may be sent at default priority.

Multicast is not supported. Each datagram must be unicast to a specific device IP.

The device replies to the source address and port from which the originating packet was received. Hosts **must** bind a fixed ephemeral port for the streaming channel for the duration of a session, and **must not** change source port mid-stream — doing so will silently strand the per-packet status replies.

Default socket buffer sizes

On the device, UDP receive buffers are tuned for high-rate streaming. The streaming socket accepts datagrams up to **1520 bytes** (the LWIP receive buffer size). Larger datagrams will be dropped without notification.

3. Discovery Channel — UDP/45456

The discovery channel is a one-byte request, one-shot reply endpoint used to detect the presence of a WiFiCube on the network and (optionally) to obtain its globally-unique identifier (the MAC address of the WiFi interface).

Request: PING (1 byte)

Offset	Size	Description
0	uint8	`0x27` — `LASERCUBE_GET_ALIVE`

Response (2 bytes)

Offset	Size	Description
0	uint8	`0x27` — echo of request opcode
1	uint8	`0x00` — alive status (always 0)

Request: PING_WITH_ID (1 byte)

Offset	Size	Description
0	uint8	`0x28` — `LASERCUBE_GET_ALIVE_WTH_ID`

Response (7 bytes)

Offset	Size	Description
0	uint8	`0x28` — echo of request opcode
1..6	uint8[6]	48-bit MAC address (efuse-programmed, MSB first as transmitted by the firmware). Treat this as the device's persistent unique identifier.

*Discovery requests are stateless. A device replies regardless of the streaming or DAC state. Because UDP is unreliable, hosts **should** retry the ping at least three times with a short timeout (≈100 ms) before declaring a device offline.*

4. Command Channel — UDP/45457

4.1 Request and response framing

Every command datagram begins with a one-byte opcode. Subsequent bytes are command-specific arguments. The maximum accepted command length is `64 bytes`.

For most commands the device returns a fixed 2-byte response of the form **[opcode, status]**, where **status** is `0x00` on success and `0x01` on failure. Several commands return larger structured payloads; these are documented individually below. A command whose opcode is unknown to the firmware returns `[opcode, 0x01]`.

The device processes commands serially from a single task. Hosts may pipeline commands but **should not** assume out-of-order responses; the n-th reply corresponds to the n-th request from the same source endpoint.

4.2 Command reference

Opcode	Mnemonic	Args	Reply	Description
0x77	GET_FULL_INFO	—	64 B	Returns a single-shot snapshot of firmware, status, DAC, and identity fields. See § 4.3.
0x78	ENABLE_BUFFER_RESPONSE	1 B (0/1)	2 B	Enable/disable the per-packet buffer-status reply on the streaming channel. Default is enabled.
0x80	SET_OUTPUT	1 B (0/1)	2 B	`0x01` enables DAC output, `0x00` disables it. Note that streaming a valid sample packet to UDP/45458 will auto-enable output (§ 5.5).
0x82	SET_ILDA_RATE	uint32 LE (pps)	2 B	Sets the DAC sample-output rate in points per second. Clamped to the device's maximum rate (see GET_FULL_INFO).
0x8A	GET_RING_FREE	—	4 B	Returns ringbuffer free-slot count. Reply: `[0x8A, 0x00, free_lo, free_hi]`. The same value is delivered automatically with each streaming packet (§ 5.3).
0x8D	CLEAR_RINGBUFFER	—	2 B	Discards all queued samples; useful before starting a new show.
0xA0	SET_DAC_BUF_THOLD_LVL	uint32 LE	2 B	Sets the target ringbuffer fill level used by the device's adaptive-rate algorithm (samples).
0xB0	SECURITY_REQUEST	≥ 10 B	2 B (deferred)	Initiates a hardware-secured operation. Reply payload is fetched separately via 0xB1.
0xB1	SECURITY_RESPONSE	—	64 B	Returns the result of the most recent SECURITY_REQUEST. Layout is product-specific.

Opcodes not listed in the table above are reserved. The firmware will echo the opcode and return status `0x01 (FAIL)`.

4.3 GET_FULL_INFO response (64 bytes)

This is the canonical way to obtain device state in a single round trip. The reply is always exactly 64 bytes; trailing bytes after the model name are zero-padded.

Offset	Size	Field
0	uint8	Opcode echo (`0x77`)
1	uint8	Status (`0x00` = success)
2	uint8	Message version (currently 0)
3	uint8	Firmware major version
4	uint8	Firmware minor version
5	uint8	Status byte. Bit 0 = output enabled; bit 1 = interlock enabled; bit 2 = temperature warning; bit 3 = over-temperature fault; bits 4–7 = packet-error count saturating at 15.

Offset	Size	Field
6..7	uint16 LE	DAC minimum value (`0x0000`)
8..9	uint16 LE	DAC maximum value (`0x0FFF`, 12-bit full scale)
10..13	uint32 LE	Current ILDA rate (samples per second)
14..17	uint32 LE	Maximum ILDA rate (samples per second)
18	uint8	Sample element count (3 = R, G, B colour channels)
19..20	uint16 LE	Ringbuffer free samples (current)
21..22	uint16 LE	Ringbuffer total capacity (6000)
23	uint8	Battery remaining percentage. Values 0–100 represent the normal state-of-charge in percent. The sentinel value `255` (`0xFF`) means the device is currently charging — hosts should display this as a charging indicator rather than as a percentage.
24	int8	Internal temperature in degrees Celsius (signed two's-complement, –128..+127). The firmware transmits this byte as a `uint8` for convenience; hosts MUST reinterpret it as a signed 8-bit value to handle sub-zero readings correctly.
25	uint8	Connection mode: 0 = LAN server, 1 = WiFi server, 2 = LAN client, 3 = WiFi client
26..31	uint8[6]	Unique device ID (MAC address)
32..35	uint8[4]	Device IPv4 address
36	uint8	Model region code
37	uint8	Model type number
38..62	char[25]	Null-terminated model name (ASCII, ≤ 25 chars + terminator)
63	uint8	Reserved, always `0x00` (null terminator safety)

5. Streaming Channel — UDP/45458

This is the high-rate path on which the host pushes XY/RGB samples to the device's DAC ringbuffer. The device feeds the ringbuffer to a hardware SPI DAC at a fixed sample rate set with `SET_ILDA_RATE` (default 35000 pps).

5.1 Data packet format

Each datagram carries a 4-byte header followed by zero or more fixed-size samples. The total datagram length must satisfy **length = 4 + 10 × N** with N ≥ 0.

Offset	Size	Field
0	uint8	Magic / packet-type byte. Must be `0xA9` (`LASERCUBE_SAMPLE_DATA_ID`) for uncompressed samples. Any other value causes the firmware to log *"Data RX unknown pkt"* and discard the datagram.
1	uint8	Reserved. Set to `0x00` on transmit.
2	uint8	Running number (sequence counter). Increments by 1 modulo 256 with each sample-bearing packet sent. See § 5.4.

Offset	Size	Field
3	uint8	Reserved. Set to `0x00` on transmit.
4..(4+10·N–1)	dac_sample_t × N	Zero or more samples laid out contiguously, each 10 bytes (see § 5.2).

With the device's 1520-byte receive buffer, the firmware will accept up to **151** samples per datagram. To stay within the standard 1500-byte Ethernet MTU and avoid IP fragmentation, hosts **should** limit themselves to **146** samples per datagram (this leaves room for the 20-byte IPv4 header and 8-byte UDP header).

> *Datagrams whose payload length is shorter than 4 bytes, or whose post-header byte count is not a multiple of 10, are treated as malformed and dropped, with the per-packet error counter incremented.*

5.2 Sample format (dac_sample_t)

Each sample is a packed structure of five little-endian 16-bit fields, totalling 10 bytes:

Offset	Size	Field	Range	Notes
0..1	uint16 LE	X	`0x000..0xFFF`	Galvo X position. 12 bits, mid-scale = 0x800.
2..3	uint16 LE	Y	`0x000..0xFFF`	Galvo Y position. 12 bits, mid-scale = 0x800.
4..5	uint16 LE	R	`0x000..0xFFF`	Red channel intensity, 12 bits.
6..7	uint16 LE	G	`0x000..0xFFF`	Green channel intensity, 12 bits.
8..9	uint16 LE	B	`0x000..0xFFF`	Blue channel intensity, 12 bits.

All channel values use only the lower 12 bits (the underlying DAC is a `DAC128S085`, 12-bit). The upper four bits of each uint16 are ignored by the firmware; hosts should clear them for forward compatibility.

Reference C definition (from *dac_control/dac_ctrl.h*):

```
typedef struct {
    uint16_t x;   // 12-bit, 0x000..0xFFF, centre 0x800
    uint16_t y;   // 12-bit, 0x000..0xFFF, centre 0x800
    uint16_t r;   // 12-bit, 0x000..0xFFF
    uint16_t g;   // 12-bit, 0x000..0xFFF
    uint16_t b;   // 12-bit, 0x000..0xFFF
} dac_sample_t;
```

5.3 Per-packet buffer status response

By default, after every successfully-decoded sample-bearing datagram the firmware sends a 4-byte status reply to the originating (IP, port):

Offset	Size	Description
0	uint8	`0x8A` — `LASERCUBE_CMD_GET_RINGBUFFER_EMPTY_SAMPLE_COUNT`
1	uint8	Echo of the running number from the data packet that triggered this reply
2..3	uint16 LE	Ringbuffer free-slot count after this packet has been queued

Hosts use the running-number echo to correlate replies with sent packets and the free-slot count to pace transmission (§ 6). The reply is best-effort UDP and may itself be lost.

Replies can be disabled with command `0x78` (`ENABLE_BUFFER_RESPONSE`, payload `0x00`). Unless the host has another flow-control mechanism in place this is not recommended.

5.4 Running number semantics

The running number (rnum) is an 8-bit counter the host increments by exactly 1 for every sample-bearing data packet, wrapping naturally from 255 to 0. It serves three purposes:

Loss detection. The firmware tracks the expected next rnum. If the received rnum is neither the expected value nor the previous value, an internal packet-loss counter is incremented and observable via `GET_FULL_INFO`.

Duplicate suppression. If the received rnum equals the rnum of the immediately-preceding packet, the firmware silently drops the duplicate (no buffering, no error increment).

Reply correlation. The status reply (§ 5.3) carries the received rnum back unchanged, allowing the host to match each reply to the packet that produced it.

When the host is starting (or restarting) a stream, the first rnum value may be any byte. The firmware self-synchronises on the first packet after the DAC transitions from off to on; subsequent packets must increment monotonically.

5.5 Auto DAC enable

If the device receives a valid sample-bearing data packet while the DAC is disabled, the firmware automatically enables output (equivalent to `SET_OUTPUT 0x01`) and resets internal packet error counters. Hosts therefore **do not need to send** `SET_OUTPUT` as part of their start-up sequence.

Conversely, the host should send `SET_OUTPUT 0x00` (and ideally `CLEAR_RINGBUFFER`) at the end of a session to ensure the laser turns off cleanly even if the host process exits.

5.6 Validation, errors, and counters

The firmware applies the following checks per datagram:

Length sanity: if length < 4, the datagram is logged as *“invalid UDP”* and discarded.

Magic byte: if byte 0 ≠ `0xA9`, the datagram is logged as *“Data RX unknown pkt”* and discarded.

Sample alignment: if (length – 4) mod 10 ≠ 0, the datagram is treated as malformed and the per-packet error counter is incremented (saturates at 255).

Buffer overflow: if the ringbuffer cannot accept all samples in the datagram, the surplus is dropped and an *overflow event* is latched (visible to UI subsystems and via the per-packet error counter).

Duplicate rnum: silently dropped (no buffering, no error).

Skipped rnum: the device's packet-loss event counter is incremented but the packet itself is still buffered.

6. Flow Control and Pacing

The device's hardware-paced DAC drains the ringbuffer at a fixed rate. The host's job is to keep the ringbuffer at a target fill level — full enough to absorb transient network jitter, not so full that you waste latency or risk overflow.

Recommended buffer target

Hosts **should aim to keep approximately** `1500 samples` **queued in the device ringbuffer** — i.e. drive the *used* count toward 1500 (or equivalently the *free-slot* count, returned in the per-packet status reply, toward 4500 = `4500`). This level has been found to give the best network stability across both Wi-Fi and wired Ethernet links: it is large enough to absorb the bursty jitter typical of consumer Wi-Fi, yet small enough that any transient over-fill drains within a frame or two and the DAC clock's adaptive-rate algorithm has plenty of headroom to track host-side timing drift.

DAC rate (pps)	Latency at 1500-sample buffer	Notes
20 000	75.0 ms (= 1500 / 20 000)	Lower rates (e.g. legacy ILDA content)
30 000	50.0 ms (= 1500 / 30 000)	Common ILDA rate
35,000	**42.9 ms** (= 1500 / 35,000)	Default LaserCube rate — also the device's maximum DAC rate

Worked example at the default rate of 35,000 pps: a projection that is 1500 samples behind the sender represents a one-way latency of **1500 ÷ 35,000 = 42.86 ms (≈ 43 ms)** from the instant a sample is queued by the host until the moment the galvos act on it. This is well below the human flicker-fusion threshold and is imperceptible during typical playback; interactive use cases (e.g. live drawing, MIDI-driven beam control) can drop the target lower if their network reliably supports it, but should not go below ~500 samples without explicit testing as the safety margin against jitter shrinks rapidly.

Recommended algorithm

Treat the per-packet status reply (§ 5.3) as a back-pressure signal. A simple but effective control loop:

```
// Pseudocode
TARGET_USED   = 1500                          //  ~43 ms @ 35 kpps
TARGET_FREE   = RINGBUFFER_SAMPLES - TARGET_USED   //  4500
MIN_USED      = 500                           //  hard lower bound
PKT_SAMPLES   = 140                           //  ≤ MTU
PPS           = 35000

loop:
    used = RINGBUFFER_SAMPLES - last_free
    samples_to_send = min(PKT_SAMPLES,
                          TARGET_USED - used + PKT_SAMPLES)
    if samples_to_send <= 0:
        sleep(PKT_SAMPLES * 1e6 / PPS); continue
    send_data_packet(rnum++, samples_to_send)
    reply = recv_status_reply(timeout=20ms)
    if reply: last_free = reply.free_slots
    if used < MIN_USED:    send_immediately()
    else if used >= TARGET_USED: sleep(samples_to_send * 1e6 / PPS)
    else:                  sleep((samples_to_send * 1e6 / PPS) / 2)
```

In practice, sending in fixed-size chunks of **146** samples (1472 bytes total) at a cadence that matches the configured ILDA rate works well. The target ringbuffer fill level can also be set explicitly via `SET_DAC_BUF_THOLD_LVL` (opcode 0xA0); the device's internal adaptive-rate algorithm uses this value to nudge the DAC clock by ±1 LSB and absorb host clock drift.

Common pitfalls

Bursting too hard at start-up. Until the device has buffered a few hundred samples its adaptive clock has nothing to lock onto. Send the first ~50 ms of stream at slightly above PPS (e.g. PPS×1.05) to fill the ringbuffer to the target level, then drop to nominal cadence.

Ignoring lost replies. If a status reply does not arrive within the round-trip budget, do not block — treat it as a hint that the network is congested and back off slightly. Replies are best-effort.

Source-port drift. NAT or firewall middleboxes that reassign your source port mid-stream will silently break the reply path. Bind to a fixed ephemeral port for the lifetime of the session.

7. Constants and Limits

Symbol	Value	Notes
`UDP_ALIVE_PORT`	45456	Discovery
`UDP_CMD_PORT`	45457	Command channel
`UDP_DATA_PORT`	45458	Streaming channel
`LASERCUBE_SAMPLE_DATA_ID`	`0xA9`	Data packet magic byte
`LASERCUBE_RINGBUFFER_SAMPLES`	6000	Total ringbuffer capacity (samples)
`LASERCUBE_DEFAULT_ILDA_RATE`	35000 pps	Boot-time DAC sample rate
`LASERCUBE_RESPONSE_SIZE_BYTES`	64	Fixed size for structured command replies
`DAC128S085_DAC_VAL_MIN`	`0x000`	DAC minimum code (12-bit)
`DAC128S085_DAC_VAL_MAX`	`0xFFF`	DAC maximum code (12-bit)
`LASERCUBE_ILDA_CHANNELS`	3	Colour channels per sample (R, G, B)
Sample size	10 bytes	Five little-endian uint16 fields
Data header size	4 bytes	Magic + reserved + rnum + reserved
Max accepted datagram	1520 bytes	Device LWIP RX buffer
Recommended max datagram	1472 bytes	Largest unfragmented IPv4/UDP payload over standard Ethernet
Max samples / datagram (firmware)	151	(1520 – 4) ÷ 10
Max samples / datagram (recommended)	146	(1472 – 4) ÷ 10

8. Worked Example — Reference Sender

The Python program below is a complete, real-device-tested reference sender. It performs discovery, calls `GET_FULL_INFO`, configures the DAC rate, opens a streaming session, continuously projects a full-screen white circle with closed-loop pacing against the per-packet status replies, prints per-second telemetry, and on exit (Ctrl-C / SIGTERM / unhandled exception) sends `CLEAR_RINGBUFFER` and `SET_OUTPUT 0` so the laser is never left emitting. It is included verbatim — copy it into a `.py` file and run it as `python3 circle_sender.py 192.168.4.1` (or whatever the device address is). Run with `--help` to see all CLI options.

> ***Why a circle and not a single point?*** *A static beam — successive samples sharing the same X and Y — concentrates laser energy at one spot, producing a dangerously bright dot ('hot beam') and risking thermal damage to optics, surfaces, and (most importantly) eyes. It also stalls the galvos at one position, defeating the protective effect of their natural mechanical motion. A continuously moving figure such as a circle distributes the laser dwell time across many points and keeps the galvos in motion at all times.* ***Hosts must never project static or near-static patterns at full intensity.***

Galvo inertia. Galvanometer scanners cannot follow arbitrarily large per-sample steps; the moving mass takes time to accelerate. For a circle of radius R drawn with N samples per revolution at P samples per second the per-sample chord length is approximately 2π·R / N DAC counts and the rotational rate is P / N revolutions per second. The example uses a full-projection radius (`0x7FF`), 600 samples per revolution at 35 000 pps — ≈ 58 rev/s and a per-sample step of ≈ 21 counts. Reducing N or increasing R without re-checking the resulting linear velocity will produce a visibly distorted (lagged, rounded-corner) projection.

```
#!/usr/bin/env python3
"""
WiFiCube Mk2 — UDP DAC streaming sanity tester.

Streams a continuous white circle at the device's DAC rate and prints
flow-control telemetry from the per-packet status replies. Acts as a
real-device validation of the protocol described in:

    docs/WiFiCube_Mk2_UDP_DAC_Streaming_Protocol_Spec.pdf

Typical use (device in WiFi-AP mode at its default address):

    python3 circle_sender.py 192.168.4.1

Press Ctrl-C to stop — the script always sends CLEAR_RINGBUFFER (0x8D)
and SET_OUTPUT 0 (0x80, 0x00) on exit so the laser is never left on.

Safety: the example deliberately projects a moving circle, never a
static point. Do not trivially modify it to stream a single (x, y)
forever — see § 8 of the spec for the rationale.
"""

from __future__ import annotations

import argparse
import math
import signal
import socket
import struct
import sys
import time

# --- Protocol constants (from the spec) ----------------------------------
PORT_ALIVE = 45456
PORT_CMD   = 45457
PORT_DATA  = 45458

MAGIC_DATA            = 0xA9
RESERVED              = 0x00
OPC_ALIVE             = 0x27
OPC_ALIVE_WITH_ID     = 0x28
OPC_GET_FULL_INFO     = 0x77
OPC_SET_OUTPUT        = 0x80
OPC_SET_ILDA_RATE     = 0x82
OPC_GET_RING_FREE     = 0x8A
OPC_CLEAR_RINGBUFFER  = 0x8D

DAC_MIN, DAC_MAX = 0x000, 0xFFF
DAC_CENTER       = 0x800

# --- CLI -----------------------------------------------------------------
def parse_args() -> argparse.Namespace:
    p = argparse.ArgumentParser(
        description="Stream a white circle to a WiFiCube Mk2 over UDP.",
```

```
        formatter_class=argparse.ArgumentDefaultsHelpFormatter,
    )
    p.add_argument("target", help="Device IPv4 address (e.g. 192.168.4.1)")
    p.add_argument("--rate", type=int, default=35000,
                   help="DAC sample rate in points per second")
    p.add_argument("--radius", type=lambda s: int(s, 0), default=0x7FF,
                   help="Circle radius in DAC counts (0..0x7FF). Default "
                        "0x7FF fills the full projection area")
    p.add_argument("--points", type=int, default=600,
                   help="Samples per revolution (smaller -> faster, "
                        "larger -> smoother, watch galvo inertia)")
    p.add_argument("--samples-per-packet", type=int, default=140,
                   help="Number of samples per UDP datagram (≤ 146 keeps "
                        "the datagram inside a 1500-byte Ethernet MTU)")
    p.add_argument("--brightness", type=lambda s: int(s, 0), default=0xFFF,
                   help="White intensity per channel, 0..0xFFF")
    p.add_argument("--no-set-rate", action="store_true",
                   help="Skip sending SET_ILDA_RATE — use whatever rate "
                        "the device is already configured for")
    p.add_argument("--duration", type=float, default=0.0,
                   help="Auto-stop after N seconds (0 = run until Ctrl-C)")
    p.add_argument("--quiet", action="store_true",
                   help="Suppress per-second status output")
    return p.parse_args()


# --- Helpers -----------------------------------------------------------------
def alive_ping(ip: str, timeout: float = 0.5) -> tuple[bool, bytes | None]:
    """Send an alive-with-ID ping. Returns (ok, mac_bytes_or_None)."""
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.settimeout(timeout)
    try:
        s.sendto(bytes([OPC_ALIVE_WITH_ID]), (ip, PORT_ALIVE))
        reply, _ = s.recvfrom(16)
        if len(reply) >= 7 and reply[0] == OPC_ALIVE_WITH_ID:
            return True, reply[1:7]
        return True, None
    except (socket.timeout, OSError):
        return False, None
    finally:
        s.close()


def get_full_info(ip: str, timeout: float = 1.0) -> dict | None:
    """Issue GET_FULL_INFO and parse the 64-byte response."""
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.settimeout(timeout)
    try:
        s.sendto(bytes([OPC_GET_FULL_INFO]), (ip, PORT_CMD))
        reply, _ = s.recvfrom(128)
        if len(reply) < 64 or reply[0] != OPC_GET_FULL_INFO:
            return None
        battery = reply[23]
        return {
            "fw_major":      reply[3],
            "fw_minor":      reply[4],
            "output_on":     bool(reply[5] & 0x01),
            "interlock":     bool(reply[5] & 0x02),
            "temp_warn":     bool(reply[5] & 0x04),
            "temp_fault":    bool(reply[5] & 0x08),
            "pkt_err_count": (reply[5] >> 4) & 0x0F,
            "current_pps":   int.from_bytes(reply[10:14], "little"),
            "max_pps":       int.from_bytes(reply[14:18], "little"),
            "ring_free":     int.from_bytes(reply[19:21], "little"),
            "ring_total":    int.from_bytes(reply[21:23], "little"),
            "battery":       battery,
            "battery_str":   "charging" if battery == 0xFF else f"{battery}%",
            "temperature":   struct.unpack("b", reply[24:25])[0],  # int8
            "conn_mode":     reply[25],
            "mac":           ":".join(f"{b:02x}" for b in reply[26:32]),
            "ip":            ".".join(str(b) for b in reply[32:36]),
            "model_name":    reply[38:63].split(b"\x00", 1)[0]
                             .decode("ascii", errors="replace"),
        }
```

```
    except (socket.timeout, OSError):
        return None
    finally:
        s.close()


def send_cmd(ip: str, payload: bytes) -> None:
    """Fire-and-forget command on UDP/45457."""
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    try:
        s.sendto(payload, (ip, PORT_CMD))
    finally:
        s.close()


def clear_ringbuffer(ip: str) -> None:
    send_cmd(ip, bytes([OPC_CLEAR_RINGBUFFER]))


def set_output(ip: str, on: bool) -> None:
    send_cmd(ip, bytes([OPC_SET_OUTPUT, 0x01 if on else 0x00]))


def set_ilda_rate(ip: str, pps: int) -> None:
    send_cmd(ip, bytes([OPC_SET_ILDA_RATE]) + struct.pack("<I", pps))


def build_circle(center: int, radius: int, points: int,
                 brightness: int) -> list[bytes]:
    """Pre-compute a full revolution as a list of packed dac_sample_t bytes."""
    radius = max(0, min(radius, DAC_CENTER - 1))
    brightness = max(0, min(brightness, DAC_MAX))
    samples = []
    for i in range(points):
        a = 2 * math.pi * i / points
        x = max(DAC_MIN, min(DAC_MAX, center + int(radius * math.cos(a))))
        y = max(DAC_MIN, min(DAC_MAX, center + int(radius * math.sin(a))))
        samples.append(
            struct.pack("<HHHHH", x, y, brightness, brightness, brightness)
        )
    return samples


# --- Main streaming loop ---------------------------------------------------
class Streamer:
    def __init__(self, args: argparse.Namespace) -> None:
        self.args = args
        self.circle = build_circle(DAC_CENTER, args.radius, args.points,
                                   args.brightness)
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.bind(("", 0))            # fixed source port
        # Non-blocking: we drain status replies between sends without ever
        # waiting on them. Blocking on recv was the original throughput bug —
        # at 4 ms per packet a 20 ms timeout caps the loop at ~50 Hz.
        self.sock.setblocking(False)
        self.running = True
        self.rnum = 0
        self.idx = 0
        # Target ringbuffer fill (samples). Keeps ~43 ms latency at 35 kpps
        # and gives plenty of network-jitter headroom — see § 6 of the spec.
        self.target_used = 1500
        self.stats = {
            "packets_sent": 0,
            "replies": 0,
            "no_reply_loops": 0,
            "rnum_mismatches": 0,
            "min_free": None,
            "max_free": None,
            "last_free": None,
        }

    def stop(self, *_):
        self.running = False
```

```
def _next_payload(self) -> bytes:
    end = self.idx + self.args.samples_per_packet
    if end <= self.args.points:
        chunk = self.circle[self.idx:end]
    else:
        chunk = self.circle[self.idx:] + self.circle[:end - self.args.points]
    self.idx = end % self.args.points
    return b"".join(chunk)

def _drain_replies(self) -> None:
    """Pull every status reply waiting in the kernel buffer."""
    while True:
        try:
            reply, _ = self.sock.recvfrom(16)
        except BlockingIOError:
            return
        if len(reply) >= 4 and reply[0] == OPC_GET_RING_FREE:
            free = reply[2] | (reply[3] << 8)
            self.stats["replies"] += 1
            self.stats["last_free"] = free
            mn, mx = self.stats["min_free"], self.stats["max_free"]
            self.stats["min_free"] = free if mn is None else min(mn, free)
            self.stats["max_free"] = free if mx is None else max(mx, free)

def _send_one(self, ip: str) -> None:
    payload = self._next_payload()
    header = bytes([MAGIC_DATA, RESERVED, self.rnum & 0xFF, RESERVED])
    try:
        self.sock.sendto(header + payload, (ip, PORT_DATA))
    except OSError as e:
        print(f"send error: {e}", file=sys.stderr)
        time.sleep(0.1)
        return
    self.stats["packets_sent"] += 1
    self.rnum = (self.rnum + 1) & 0xFF

def run(self) -> None:
    ip = self.args.target
    pkt = self.args.samples_per_packet
    pps = self.args.rate
    period = pkt / pps           # nominal seconds per packet

    # ---- Phase 1: pre-fill burst ----------------------------------------
    # Spray packets back-to-back until the device reports the ringbuffer
    # has reached target_used. Cap the burst so a missing reply (e.g. the
    # device hasn't started DAC yet) can't spin forever — at 1500 samples
    # / 140 per packet that's ~11 packets; we allow a generous 40.
    if not self.args.quiet:
        print(f"phase 1: pre-filling buffer to {self.target_used} samples...")
    for _ in range(40):
        if not self.running:
            break
        self._send_one(ip)
        # Brief sleep so we don't exceed the device's RX socket queue
        time.sleep(0.001)
        self._drain_replies()
        free = self.stats["last_free"]
        if free is not None and (6000 - free) >= self.target_used:
            break

    # ---- Phase 2: steady state ------------------------------------------
    if not self.args.quiet:
        print("phase 2: steady-state streaming")
    next_send = time.monotonic()
    next_status = time.monotonic() + 1.0
    deadline = (time.monotonic() + self.args.duration
                if self.args.duration > 0 else None)

    while self.running:
        if deadline and time.monotonic() >= deadline:
            break

        self._send_one(ip)
        self._drain_replies()
```

```
            # Adaptive pacing.
            # If the device tells us the buffer is below target, send extra
            # packets at zero pacing until we catch up. If above target, just
            # tick at the nominal period.
            free = self.stats["last_free"]
            if free is not None:
                used = 6000 - free
                if used < self.target_used - pkt:
                    # We're behind — re-baseline so we don't sleep
                    next_send = time.monotonic()
                # else: stay on the nominal cadence
            else:
                # Haven't received any reply yet — assume behind, send fast
                self.stats["no_reply_loops"] += 1
                next_send = time.monotonic()

            # Status line once per second
            if not self.args.quiet and time.monotonic() >= next_status:
                self._print_status(pps)
                next_status += 1.0

            # Pace
            next_send += period
            delay = next_send - time.monotonic()
            if delay > 0:
                time.sleep(delay)
            elif delay < -0.05:
                # Significantly behind schedule — re-baseline so we don't
                # accumulate a debt that turns into a flood.
                next_send = time.monotonic()

        if not self.args.quiet:
            self._print_status(pps, final=True)

    def _print_status(self, pps: int, final: bool = False) -> None:
        free = self.stats["last_free"]
        used = (None if free is None else 6000 - free)
        latency_ms = (None if used is None else used * 1000.0 / pps)
        prefix = "final " if final else ""
        free_s = f"{free:>5}" if free is not None else "    -"
        used_s = f"{used:>5}" if used is not None else "    -"
        lat_s  = f"{latency_ms:6.1f}ms" if latency_ms is not None else "   -- "
        print(
            f"[{prefix}status] "
            f"sent={self.stats['packets_sent']:>7d}  "
            f"replies={self.stats['replies']:>6d}  "
            f"no-reply-loops={self.stats['no_reply_loops']:>4d}  "
            f"free={free_s}  "
            f"used={used_s}  "
            f"latency≈{lat_s}",
            flush=True,
        )


# --- Entry point -------------------------------------------------------------
def main() -> int:
    args = parse_args()

    # 1) Liveness check
    ok, mac = alive_ping(args.target)
    if not ok:
        print(f"error: no alive reply from {args.target}:{PORT_ALIVE} — "
              "is the device powered on and reachable?", file=sys.stderr)
        return 2
    if mac:
        mac_str = ":".join(f"{b:02x}" for b in mac)
        print(f"device alive at {args.target}  (MAC {mac_str})")
    else:
        print(f"device alive at {args.target}")

    # 2) Identify
    info = get_full_info(args.target)
    if info:
        print(f"  firmware:    {info['fw_major']}.{info['fw_minor']}")
```

```
        print(f"  model:       {info['model_name'] or '(unset)'}")
        print(f"  current pps: {info['current_pps']}  "
              f"(max {info['max_pps']})")
        print(f"  ring:        {info['ring_total']} samples total, "
              f"{info['ring_free']} free")
        print(f"  battery:     {info['battery_str']}")
        print(f"  temperature: {info['temperature']} °C")
        flags = []
        if info["output_on"]:   flags.append("output ON")
        if info["interlock"]:   flags.append("interlock OK")
        if info["temp_warn"]:   flags.append("TEMP WARNING")
        if info["temp_fault"]:  flags.append("TEMP FAULT")
        print(f"  flags:       {', '.join(flags) or '(none)'}")
        if info["pkt_err_count"]:
            print(f"  pkt errors:  {info['pkt_err_count']} (will reset on stream start)")

    # 3) Configure DAC rate (unless user opted out)
    if not args.no_set_rate:
        set_ilda_rate(args.target, args.rate)
        time.sleep(0.05)

    # 4) Stream
    streamer = Streamer(args)
    signal.signal(signal.SIGINT,  streamer.stop)
    signal.signal(signal.SIGTERM, streamer.stop)
    print(f"streaming circle: r=0x{args.radius:03X}  "
          f"points/rev={args.points}  pkt={args.samples_per_packet}  "
          f"rate={args.rate} pps  "
          f"(target latency at 1500-sample fill: "
          f"{1500 * 1000.0 / args.rate:.1f} ms)")
    print("press Ctrl-C to stop")
    try:
        streamer.run()
    finally:
        # Always leave the laser in a safe state
        clear_ringbuffer(args.target)
        time.sleep(0.02)
        set_output(args.target, False)
        print("\nclean shutdown: CLEAR_RINGBUFFER + SET_OUTPUT 0 sent.")

    return 0


if __name__ == "__main__":
    sys.exit(main())
```

Two details in the script above are protocol-critical and easy to get wrong on the first attempt. **The receive socket is non-blocking** — a blocking recv with even a small timeout caps the loop frequency to roughly the timeout reciprocal and starves the DAC of samples. Status replies are drained in a tight `BlockingIOError`-bounded loop that returns immediately when the kernel queue is empty (the equivalent on other stacks is `EWOULDBLOCK / EAGAIN` or `WSAEWOULDBLOCK`). **The startup phase sprays packets back-to-back** until the device confirms the ringbuffer has reached the 1500-sample target, only then falling into nominal pacing. Skipping the pre-fill burst means the DAC drains faster than the host delivers for the first several hundred milliseconds, producing visible flicker or underruns until the loop catches up.

Shutdown is mandatory. The `try / finally` block guarantees that `CLEAR_RINGBUFFER` (0x8D) and `SET_OUTPUT 0` (0x80, 0x00) are sent on every exit path — normal completion, signals, or uncaught exceptions. Production hosts must also catch SIGINT/SIGTERM explicitly and call into the same cleanup, so that the laser is never left emitting if the host process is killed externally.

A production sender will additionally: discover the device via the alive port (UDP/45456), call `GET_FULL_INFO` (0x77) at startup to confirm firmware version, current/maximum ILDA rate, and warning/fault flags (in particular interlock state and over-temperature), set the desired DAC rate with `SET_ILDA_RATE` (0x82), surface telemetry from the per-packet replies to the operator, and ramp the white intensity from black across the first revolution to give the galvos time to settle onto the path before full

output.

9. Implementation Checklist

- Bind a single fixed UDP source port for the streaming socket and reuse it for the entire session.
- Mark streaming traffic with DSCP CS4/AF41 (TOS 0x80–0xA0) for WMM Voice/Video class on Wi-Fi networks.
- Limit datagrams to ≤ 146 samples (1472 bytes total) on standard-MTU networks.
- Increment the running number by exactly 1 per sample-bearing packet; never repeat a value.
- Treat the per-packet status reply as advisory; tolerate loss of individual replies without stalling the sender.
- Pre-fill the ringbuffer to the target threshold before expecting the device to track your stream tightly.
- On stream end, send `CLEAR_RINGBUFFER` (0x8D) and `SET_OUTPUT 0` (0x80, 0x00) to leave the laser in a safe state.
- Periodically call `GET_FULL_INFO` (0x77) to surface battery, temperature, and packet-error metrics in your UI.
- Use the 6-byte unique ID (MAC) from the alive-with-id reply, not the IP address, to identify a device persistently.

Appendix A — Endianness and Byte Order

Every multibyte integer described in this specification is transmitted in **little-endian** order, matching the ESP32-S3 native CPU byte order. Hosts implemented on big-endian platforms (rare in 2026 but possible on some embedded SoCs and historic networking gear) **must** byte-swap before transmission.

The discovery-with-ID response (§ 3) carries the MAC address in the natural transmit order used by IEEE 802 — that is, the first byte on the wire is the OUI's most significant octet. It is *not* a little-endian uint64.

Appendix B — Diagnostics

To inspect traffic on a running system, use Wireshark with the display filter `udp.port == 45458` for streaming or `udp.port == 45457` for control. Each streaming datagram is easy to spot — payload starts with `0xA9` and total length is always 4 + 10 × N.

If the device appears to lose packets at scale: check that the host is not bursting more samples than the ringbuffer can absorb in one round trip; check that DSCP marking is being preserved by your AP; and call `GET_FULL_INFO` between bursts to read the saturating packet-error counter (byte 5, bits 4–7).

The firmware keeps a separate one-shot overflow flag and a one-shot loss flag, both readable internally by the device's UI subsystems but not exposed over UDP. To observe these from the host, watch for the packet-error counter incrementing in the GET_FULL_INFO snapshot.