

TECHNICAL SPECIFICATION

UDP DAC Streaming Protocol Specification

Protocol v1.0 · WiFiCube / WiFiCube Ultra MK2 / Pro

PROTOCOL VERSION	v1.0
DOCUMENT VERSION	v1.1
DATE	10 October 2026
AUTHOR	Paul Mainwaring
APPLIES TO	WiFiCube, WiFiCube Ultra MK2 and Pro (3.0W and 6.0W)
AUDIENCE	Software engineers integrating a host application that streams DAC sample data to a WiFiCube, WiFiCube Ultra MK2 or Pro over UDP
STATUS	Implementer reference — normative

Table of Contents

1. Overview	3
2. Transport and Endpoints	3
3. Discovery Channel — UDP/45456	4
4. Command Channel — UDP/45457	4
4.1 Request and response framing	4
4.2 Command reference	5
4.3 GET_FULL_INFO response (64 bytes)	5
5. Streaming Channel — UDP/45458	6
5.1 Data packet format	6
5.2 Sample format (dac_sample_t)	7
5.3 Per-packet buffer status response	7
5.4 Running number semantics	8
5.5 Auto DAC enable	8
5.6 Validation, errors, and counters	8
6. Flow Control and Pacing	9
7. Constants and Limits	10
8. Worked Example — Reference Sender	11
9. Implementation Checklist	18
Appendix A — Endianness and Byte Order	18
Appendix B — Diagnostics	18
Appendix C — Revision History	18

1. Overview

The WiFiCube, the WiFiCube Ultra MK2 and the Pro implement a three-port UDP protocol for host control and DAC streaming. The Pro comes in two models, the 3.0W Pro and the 6.0W Pro. The protocol on the wire is identical across all of them 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.

The protocol runs only in the device's LaserOS control mode. In the other control modes ports 45457 and 45458 are closed, and port 45456 answers at most the web interface's `WCUBE?1` discovery probe (in the Playlist and Visualizer modes), not `0x27` or `0x28`.

*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.

One sender at a time. While a stream is running the device takes sample packets only from the host that started it: packets from any other IP address are dropped without a reply until the streaming host has sent nothing for about one second, and the next host to send then starts a new stream. Commands on UDP/45457 are accepted from any host at any time.

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. Firmware 2.1 and earlier reply with six zero bytes here: read the MAC from GET_FULL_INFO bytes 26–31 instead.

*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. The rate must be from 6000 to the device's maximum (GET_FULL_INFO bytes 14–17). From firmware 2.2 a rate outside that range replies FAIL and the rate is unchanged; firmware 2.1 and earlier ignored it but replied success.
0x8A	GET_RING_FREE	—	4 B	Returns ringbuffer free-slot count. Reply: [0x8A, 0x00, free_lo, free_hi]. The same value comes with each streaming packet's status reply (§ 5.3), except over Wi-Fi, where that one is lower by 10 ms of samples.
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 (samples) for the device's adaptive rate, which is off by default and only acts while it is on (§ 6).
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)

Offset	Size	Field
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 = data packet errors since the previous GET_FULL_INFO, stopping at 15 (the read clears the count): skipped running numbers, overflow drops and malformed sample packets (§ 5.6). Firmware 2.1 and earlier always report 0 in bit 0 and in bits 4–7.
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 (8000)
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 $\text{length} = 4 + 10 \times N$ with $N \geq 0$.

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

Offset	Size	Field
		<i>RX unknown pkt</i> ” 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.
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).

A datagram shorter than 4 bytes is dropped without counting an error. A sample packet whose post-header byte count is not a multiple of 10 is dropped whole and counts as one error (GET_FULL_INFO byte 5); firmware 2.1 and earlier still played its complete samples. Hosts should only ever send 4 + 10 × N bytes.

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..0xFFFF	Galvo X position. 12 bits, mid-scale = 0x800.
2..3	uint16 LE	Y	0x000..0xFFFF	Galvo Y position. 12 bits, mid-scale = 0x800.
4..5	uint16 LE	R	0x000..0xFFFF	Red channel intensity, 12 bits.
6..7	uint16 LE	G	0x000..0xFFFF	Green channel intensity, 12 bits.
8..9	uint16 LE	B	0x000..0xFFFF	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..0xFFFF, centre 0x800
    uint16_t y; // 12-bit, 0x000..0xFFFF, centre 0x800
    uint16_t r; // 12-bit, 0x000..0xFFFF
    uint16_t g; // 12-bit, 0x000..0xFFFF
    uint16_t b; // 12-bit, 0x000..0xFFFF
} 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_COU

Offset	Size	Description
NT		
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.

Over Wi-Fi the device subtracts 10 ms of samples at the current rate from this free-slot count, to allow for the packets the host already has in flight, so it reads lower than GET_RING_FREE (0x8A) by that amount. Over Ethernet the two are the same.

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, the packet-error count in GET_FULL_INFO byte 5 goes up by one (from firmware 2.2), however many packets were missed, and the packet is still buffered.

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 device synchronises on the first packets after its output switches on: the first one to three are used to synchronise and are not drawn. The start of a stream is blanked anyway, so this is not visible, but a single packet sent on its own draws nothing. Subsequent packets must increment by exactly 1.

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 the packet-error count. 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 $\text{length} < 4$, the datagram is logged as “invalid UDP” and discarded.

Magic byte: if $\text{byte } 0 \neq 0xA9$, the datagram is logged as “Data RX unknown pkt” and discarded.

Sample alignment: if $(\text{length} - 4) \bmod 10 \neq 0$, the packet is dropped whole, with no status reply, and the error count goes up by one. Its running number still counts as received, so the next packet is not also counted as a gap. Firmware 2.1 and earlier counted it but still played its complete samples.

Buffer overflow: if the ringbuffer cannot accept all the samples in the datagram, the whole datagram is dropped, so the gap in the drawing has a clean edge, and the error count goes up by one.

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

Skipped rnum: the error count goes up by one (from firmware 2.2) and 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 $8000 - 1500 = 6500$). 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 host's pacing has room to correct for the drift between its clock and the device's.

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 $1500 \div 35,000 = 42.86\text{ ms}$ ($\approx 43\text{ ms}$) 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 - § 8 has a complete implementation
CAPACITY      = GET_FULL_INFO bytes 21..22      // 8000
TARGET_USED   = 1500                             // ~43 ms @ 35 kpps
PKT_SAMPLES   = 140                             // <= 146 (MTU)
PPS           = 35000
PERIOD        = PKT_SAMPLES / PPS                // one packet's worth of time

every PERIOD (a fixed clock - never wait for a reply):
    drain every waiting status reply; keep the newest free count
    used = CAPACITY - last_free                   // + packets sent since that reply
                                              // - samples played since it came
    if used < TARGET_USED - PKT_SAMPLES: send 2 packets // catch up
    elif used > TARGET_USED + PKT_SAMPLES: send nothing // let it drain
    else: send 1 packet
```

In practice, sending fixed-size chunks of up to 146 samples (1464 bytes) at a cadence that matches the configured rate works well; the reference sender (§ 8) uses 140. By default nothing on the device absorbs the drift between the host's clock and its own, so the host's pacing has to correct in both directions, as

above. The device also has an optional adaptive rate, the *Enable compatibility with original cube(s)* setting in its menu, off by default: while it is on, the device steps its DAC timer by 4 counts (about 0.35 % of the rate at 35000 pps) to hold the ringbuffer near the level set with SET_DAC_BUF_THOLD_LVL (0xA0).

Common pitfalls

Starting without a pre-fill. The device holds its output, blanked, until about 1500 samples are queued (or about 100 ms after samples start arriving), so a stream starts with its full runway. Send back-to-back until the status replies show the target fill, then drop to the nominal cadence (§ 8 does this); a host that trickles in at the nominal rate from the start has no margin.

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	8000	Total ringbuffer capacity (samples)
LASERCUBE_DEFAULT_ILDA_RATE	35000 pps	Boot-time DAC sample rate
Maximum DAC rate	35000 pps	GET_FULL_INFO bytes 14–17
Minimum DAC rate	6000 pps	From firmware 2.2, SET_ILDA_RATE outside 6000 to the maximum replies FAIL
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	1464 bytes	146 samples (4 + 146 × 10). The largest unfragmented IPv4/UDP payload over standard Ethernet is 1472 bytes
Max samples / datagram (firmware)	151	$(1520 - 4) \div 10$
Max samples / datagram (recommended)	146	$(1472 - 4) \div 10$

8. Worked Example — Reference Sender

The Python program below is a complete, real-device-tested reference sender. It performs discovery, reads the ringbuffer size and maximum rate from GET_FULL_INFO, sets the DAC rate (the device's maximum unless told otherwise), 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 (including the device's packet-error count), 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 $2\pi R / N$ 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
MIN_PPS           = 6000      # lowest DAC rate the device accepts (§ 4.2)
TOS_CS4           = 0x80      # DSCP CS4: Wi-Fi WMM video queue, as the device marks its own (§ 2)

# --- 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=None,
        help="DAC sample rate in points per second (default: the "
        "device's maximum, from GET_FULL_INFO)")
    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, timeout: float = 0.5) -> bool:
    """SET_ILDA_RATE; True when the device replies SUCCESS. A rate outside
    6000..max replies FAIL and the device keeps its current rate."""
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.settimeout(timeout)
    try:
        s.sendto(bytes([OPC_SET_ILDA_RATE]) + struct.pack("<I", pps), (ip, PORT_CMD))
        reply, _ = s.recvfrom(16)
        return len(reply) >= 2 and reply[0] == OPC_SET_ILDA_RATE and reply[1] == 0x00
    except (socket.timeout, OSError):
        return False
    finally:
        s.close()

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, ring_total: int) -> None:
        self.args = args
        # Ringbuffer capacity from GET_FULL_INFO bytes 21-22, so the used
        # count is worked out from the device's own figure, not a constant.
        self.ring_total = ring_total
        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
# Mark the stream for the Wi-Fi video queue (§ 2); not every OS allows it
try:
    self.sock.setsockopt(socket.IPPROTO_IP, socket.IP_TOS, TOS_CS4)
except OSError:
    pass
# 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)
# Second socket for a once-a-second GET_FULL_INFO, also non-blocking:
# its packet-error count (byte 5, bits 4-7) clears on every read
self.cmd_sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
self.cmd_sock.setblocking(False)
self.running = True
self.rnum = 0
self.idx = 0
self.reply_used = None
self.reply_inflight = 0
self.reply_time = 0.0
# 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,
    "pkt_errors": 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
            # Snapshot for _used_now(): the fill this reply reports, the
            # packets sent after the one it answers (byte 1 echoes its
            # rnum) that it can't include yet, and when it arrived
            self.reply_used = self.ring_total - free
            self.reply_inflight = (self.rnum - reply[1] - 1) & 0xFF
            self.reply_time = time.monotonic()
            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)
    while True:
        try:
            reply, _ = self.cmd_sock.recvfrom(128)
        except BlockingIOError:
            return
        if len(reply) >= 64 and reply[0] == OPC_GET_FULL_INFO:
            # errors since the previous GET_FULL_INFO (shows at most 15)
            self.stats["pkt_errors"] += (reply[5] >> 4) & 0x0F

def _poll_errors(self, ip: str) -> None:
    try:

```

```

        self.cmd_sock.sendto(bytes([OPC_GET_FULL_INFO]), (ip, PORT_CMD))
    except OSError:
        pass

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 _used_now(self, pps: int) -> float | None:
    """Estimated ringbuffer fill now: the last reply's figure, plus the
    packets it could not include yet, less what the DAC has played since.
    Keeps the estimate moving between replies - a skipped send gets no
    reply, so the last reported figure alone would never fall."""
    if self.reply_used is None:
        return None
    played = pps * (time.monotonic() - self.reply_time)
    return (self.reply_used + self.reply_inflight * self.args.samples_per_packet
            - played)

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 -----
    # Send back-to-back until the device reports the ringbuffer has
    # reached target_used. The Ultra MK2 and Pro hold their output until
    # about that many samples are queued, so the stream starts with its
    # full runway. Capped so a missing reply (e.g. the device hasn't
    # started its 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()
        used = self._used_now(pps)
        if used is not None and used >= self.target_used:
            break

    # ---- Phase 2: steady state -----
    # A fixed clock ticks at the nominal packet period. Each tick sends
    # one packet - two while the buffer is more than a packet below the
    # target, none while it is more than a packet above. The fill level
    # holds at the target in both directions, whichever clock (host or
    # device) runs fast; without a reply yet, hold the nominal cadence.
    if not self.args.quiet:
        print("phase 2: steady-state streaming")
    next_tick = 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._drain_replies()
        used = self._used_now(pps)
        if used is None:
            self.stats["no_reply_loops"] += 1
            burst = 1
        elif used < self.target_used - pkt:
            burst = 2          # below target: catch up

```

```

        elif used > self.target_used + pkt:
            burst = 0                # above target: let it drain
        else:
            burst = 1
        for _ in range(burst):
            self._send_one(ip)

    # Status line (and the device's error count) once per second
    if time.monotonic() >= next_status:
        self._poll_errors(ip)
        if not self.args.quiet:
            self._print_status(pps)
        next_status += 1.0

    # Pace
    next_tick += period
    delay = next_tick - time.monotonic()
    if delay > 0:
        time.sleep(delay)
    elif delay < -0.05:
        # The host stalled - re-baseline the clock rather than flood
        # to catch up; the fill estimate makes up the difference.
        next_tick = 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 self.ring_total - 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"snt={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} "
        f"pkt-errors={self.stats['pkt_errors']}",
        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']} since the last GET_FULL_INFO")

# 3) Rate and buffer size from the device (GET_FULL_INFO), not constants
max_pps = info["max_pps"] if info else 35000
ring_total = info["ring_total"] if info and info["ring_total"] else 8000
if args.no_set_rate and info:
    args.rate = info["current_pps"]
elif args.rate is None:
    args.rate = max_pps
if not MIN_PPS <= args.rate <= max_pps:
    print(f"error: --rate {args.rate} is outside {MIN_PPS}..{max_pps} pps "
          "for this device", file=sys.stderr)
    return 2

# 4) Configure DAC rate (unless user opted out)
if not args.no_set_rate:
    if not set_ilda_rate(args.target, args.rate):
        print(f"warning: SET_ILDA_RATE {args.rate} was not acknowledged",
              file=sys.stderr)

# 5) Stream
streamer = Streamer(args, ring_total)
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: a fixed clock that sends one packet per period, two while the buffer is below the target and none while it is above, using a fill estimate that keeps moving between replies (a skipped send gets no reply). 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 for the Wi-Fi video queue (WMM AC_VI): DSCP CS4 (TOS 0x80), as the device marks its own, or AF41 (TOS 0x88).
- Limit datagrams to ≤ 146 samples (1464 bytes) on standard-MTU networks.
- Read the ringbuffer capacity (bytes 21–22) and the maximum rate (bytes 14–17) from `GET_FULL_INFO` rather than assuming them, and keep `SET_ILDA_RATE` from 6000 to that maximum.
- 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 (firmware 2.1 and earlier reply with zeros: use `GET_FULL_INFO` bytes 26–31 there).

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 $4 + 10 \times N$.

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 packet-error count (byte 5, bits 4–7: errors since the previous read, stopping at 15; from firmware 2.2).

The device also counts lost packets for its own Status page (*PKT Loss Events*), separately from the `GET_FULL_INFO` error count: reading one does not reset the other.

Appendix C — Revision History

v1.1 — 10 October 2026 (protocol v1.0, firmware 2.2)

- Names the Pro (3.0W and 6.0W). The protocol runs only in the LaserOS control mode and takes one streaming host at a time.
- PING_WITH_ID returns the MAC from firmware 2.2 (zeros before).
- SET_ILDA_RATE outside 6000 to the maximum replies FAIL and keeps the rate (v1.0 said it was clamped).
- GET_FULL_INFO byte 5: the output bit and the error count work from firmware 2.2; what the count covers, and that the read clears it.
- Malformed sample packets (length not $4 + 10 \times N$) are dropped from firmware 2.2.
- Ringbuffer capacity is 8000. The Wi-Fi per-packet free count is 10 ms lower. Overflowing datagrams and the first packets of a stream as the firmware handles them.
- The adaptive rate is off by default and steps by 4 timer counts; hosts pace in both directions. New pseudocode.
- 146 samples are 1464 bytes; the DSCP values corrected.
- Reference sender: reads the ringbuffer size and maximum rate from the device, paces in both directions, marks its packets CS4 and shows the packet-error count.

v1.0 — 29 April 2026

- First release.