WAINLUX K6 Web abstraction on Pi Zero W

Table of Contents

Other formats: HTML | GitHub Repository

1. Summary

Open-source host software for operating Wainlux K-series laser engravers using an independently developed protocol implementation.

This project is an independently developed, clean-room implementation created through analysis of publicly observable behavior and legally obtained software. It is not affiliated with or endorsed by Wainlux.

Is:

- Host-side control software
- Protocol implementation

- Interoperability tool

Is not:

- ~~Firmware replacement~~
- ~~Vendor software redistribution~~
- ~~Reverse-engineered source release~~

Pi Zero W controls Wainlux K6 laser via USB serial. Docker containerized Flask app. Web UI. Binary protocol.

2. WARNINGS

- Laser safety: **always** wear appropriate laser safety glasses.
- Fire hazard: do not run unattended; have a fire-safe surface.
  - Keep a fire extinguisher handy, or a fire blanket.
- Use a smoke detector.

3. Wainlux K6 USB Interface

Documentation for the Wainlux K6 laser engraver control system on Pi Zero W. Includes implementation details, quick start guide, status tracking, K3 protocol reference (archived - not applicable to K6), and debug notes.

3.1. Project Status

3.1.1. How we got here

This is the shortest honest story.

1. We started with the C++ protocol code (`backup/K3_LASER_ENGRAVER_PROTOCOL/`).
2. We made it build on Linux (CMake: do not compile `win.cpp` on Linux).
3. We fixed the ACK reader in C++ (`wait_for_ack()` was lying).
4. The C++ “K3” handshake still stalled on this unit.
5. We switched to the vendor macOS Java app in `backup/` and ran it through ghidra through an LLM.
6. That revealed a different opcode set that **does** ACK and move this unit.
7. We implemented that sequence in Python and got repeatable movement and `0x09` ACKs.
8. We fumbled in the dark and had to reset the unit multiple times and listen to it push on boundaries etc.
9. We got more stable as we ran into every problem possible in no particular order.

3.1.2. Complete

- [x] Pi Zero W setup and Docker installation
- [x] USB serial driver (CP2102) working
- [x] Flask web app structure
- [x] Docker container builds on ARMv6
- [x] Web UI (connect, disconnect, test, engrave)
- [x] Protocol capture with vendor Windows/macOS software
- [x] K6 protocol identification (differs from K3)
- [x] Basic serial communication (115200 baud)
- [x] ACK byte (0x09) confirmed
- [x] Home command works
- [x] Design and print 3D holder/case for Pi Zero with UPS and camera
- [x] Initial basic testing on actual K6 hardware
- [x] K3 protocol reverse-engineering referenced
- [x] Documentation structure established (AsciiDoc)
- [x] PlantUML diagrams for architecture and protocol
- [x] Initial test scripts (MVP vector and raster)
    - Now removed to avoid confusiuon
- [x] Fix USB device ID in Dockerfile (CP2102: 10c4:ea60)
- [x] UPS Lite integration option (power status + safe shutdown (was python2))
- [x] Camera tasks: rpicam-still capture
- [x] Java protocol extraction before deletion
- [x] Ghidra/MCP debug tooling (decompile + extract protocol details)
- [x] Job data packet format validation
- [x] Reliable burning/marking (now strong marks with correct header endianness + depth=10)
- [x] Opcode mapping (K3 vs K6 differences) - all opcodes documented
- [x] Identified failing opcode in handshake sequence (0x1c not in K6 protocol, K3-only)
- [x] Licensing implementation (MIT for code, CC-BY/CC0 for docs, clean-room statement, contributor policy)
- [x] Remove vendor java from repo (was already gitignored, now deleted)
- [x] Image size limits documentation (80mm @ 0.075mm/px = 1067x1067px max from Java)
- [x] Error handling improvements
- [x] initial vector testing on actual K6 hardware
- [x] initial image testing on actual K6 hardware
- [x] Hardware testing (raster burns with depth=10, power=1000 defaults)

- [x] Vector-only mode validation on K6
- [x] Calibrate burn completion timeout: Fixed serial timeout corruption, increased idle timeout to 90s, changed max timeout to 5× estimate with proper exit reason tracking
- [x] Library refactoring: Extracted protocol into scripts/k6 package
- [x] Transport abstraction: SerialTransport + MockTransport for testing
- [x] Exception-based error handling (K6Error, K6TimeoutError, K6DeviceError)
- [x] Unit test suite: 34 tests covering protocol, driver, CSV logging
- [x] CSV logging integration with legacy format compatibility
- [x] Flask API updated to use new transport-based driver
- [x] Fixed Flask app k6 library integration (removed placeholder, uses docker/k6/)
- [x] Added draw_bounds_transport method (no subprocess dependency)
- [x] Docker container uses pure library implementation (no legacy script)

3.1.3. In Progress

- [ ] Camera integration option (burn preview + documentation photos)
- [ ] Hardware validation with new library on Pi Zero W

3.1.4. Pending

- [ ] Adjust payload format for raster streaming (if needed after hardware testing)
- [ ] Image boundary/cropping (observed behavior: crop to work area)
- [ ] Production testing on actual K6 hardware
- [ ] Status API expansion
- [ ] Multi-job queue (if needed)
- [ ] GitHub Actions for documentation build
- [ ] PDF theme customization
- [ ] Front cover image

3.1.5. Known Issues

- Image positioning/centering can drive out of bounds (Java centering formula documented: center_x = x + width/2 + 67)
- Vector circles >20mm may go out of bounds (40mm diameter observed to go haywire)
- No persistent storage (stateless design by choice)
- Image size limit: 1067x1067px max (80mm work area @ 0.075mm/px resolution)
- Y-axis calibration: boundary test shows 80x76mm instead of expected 80x80mm (4mm shortfall)
    - need to retry

3.1.6. Architecture Decisions

- [x] Docker: Isolation and device passthrough
- [x] Flask: Lightweight (40MB less than FastAPI)
- [x] No database: 512MB RAM limit, stateless by design
- [x] Privileged container: Required for /dev/ttyUSB0
- [x] Pillow only: No OpenCV (150MB overhead)
- [x] AsciiDoc: Multi-format output, GitHub rendering
- [x] MCP/Ghidra: Reverse-engineering tools without looking at code with mine own eyes

3.1.7. Next Steps

- [ ] Y-axis calibration investigation (80x76mm vs 80x80mm)
- [ ] Implement boundary checking/cropping (prevent out-of-bounds)
- [ ] Test vector circles >20mm diameter (40mm went out of bounds)
- [ ] Test repeat count parameter (byte 36 in header)
- [ ] Validate all newly documented opcodes (0x06/07, 0x16, 0x20, 0x25, 0x28)
- [ ] Optional: Firmware update feature (IAP protocol fully documented)

3.2. Getting Started

Quick guide to using this repository.

3.2.1. For Users

3.2.1.1. Deploy to Pi

```
cd docker-wainlux
docker compose build
docker compose up -d
```

Access: `http://<pi-ip>:8080`

3.2.2. For Developers

3.2.2.1. Read Code

- K6 Driver - Protocol
- Flask App - API
- Web UI - Interface

See App README.

3.2.2.2. Generate Diagrams

```
cd images
plantuml *.puml
cd ..
```

Output: `images/*.png`

3.2.3. For Documentation Writers

3.2.3.1. Add Diagram

Create `.plantuml` file in `images/`:

```
@startuml

actor User
User --> System
@enduml
```

Generate:

```
plantuml images/new-diagram.plantuml
```

Reference in AsciiDoc:

```
\image::images/new-diagram.png[Description]
```

3.2.4. For Contributors

3.2.4.1. Clone Repository

```
git clone <repo-url> wainlux-pi
cd wainlux-pi
```

3.2.4.2. Follow Style

- DRY - Don't Repeat Yourself
- KISS - Keep It Simple, Stupid
- Hemingway - Short, direct sentences
- No fluff - Essential info only

3.2.5. Quick Commands

```
# Generate diagrams
plantuml images/*.plantuml

# Build container
cd docker-wainlux && docker compose build

# Run container
docker compose up -d

# View logs
docker compose logs -f
```

3.3. Installation and Setup

Step-by-step installation and setup for Wainlux K6 on Pi Zero W.

3.3.1. Pi Zero W setup

Initial Pi preparation for Wainlux K6.

3.3.1.1. Verified hardware

- Pi Zero W (ARMv6)
- Raspbian GNU/Linux 13 (trixie)
- 427 MiB RAM / 426 MiB swap
- Wainlux K6 (labeled K6)
- K6 via CP2102 USB-UART bridge (idVendor=10c4, idProduct=ea60)
- Device: /dev/ttyUSB0 (cp210x driver)
- Wifi

3.3.1.2. OS install

Download Raspberry Pi OS Lite

- 32-bit slim version
  - No desktop needed
- https://www.raspberrypi.com/software/

Flash with Imager (version >2)

- Enable SSH
- Configure WiFi
- Set hostname: `pi-hostname`

- User: `user` (or other)
    - Password: set your own

3.3.1.3. First boot

SSH access:

```
## for paswwordless access
#ssh-copy-id user@pi-ip
#ssh-add

ssh user@pi-ip
```

Update system:

```
sudo apt-get update && sudo apt-get upgrade -y
```

Install git:

```
sudo apt-get install -y git
```

3.3.1.4. K6 USB verification

Check USB device:

```
lsusb | grep 10c4
```

Should show:

```
Bus 001 Device 004: ID 10c4:ea60 Silicon Labs CP210x UART Bridge
```

Check serial device:

```
ls -la /dev/ttyUSB0
```

Should show:

```
crw-rw---- 1 root dialout 188, 0 Jan 11 16:16 /dev/ttyUSB0
```

Verify kernel driver:

```
dmesg | grep -i cp210x
```

Should show:

```
cp210x 1-1:1.0: cp210x converter detected
usb 1-1: cp210x converter now attached to ttyUSB0
```

3.3.2. Build instructions

Step-by-step build on Pi Zero W.

3.3.2.1. Prerequisites

1. Pi Zero W with OS
2. Docker installed
3. 1GB swap enabled
4. K6 connected via USB
5. Network access

3.3.2.2. Build

```
cd ~/wainlux-pi/docker-wainlux
docker compose build
```

Expected output:

```
[+] Building 1089.3s (12/12) FINISHED
 => [internal] load build definition
 => => transferring dockerfile
 => [internal] load .dockerignore
 => exporting to image
```

Time: 15-20 minutes.

3.3.2.3. Verify

```
docker images
```

Should show:

```
REPOSITORY              TAG       SIZE
docker-wainlux-wainlux  latest    ~280MB
```

3.3.2.4. Run

```
docker compose up -d
```

Check status:

```
docker compose ps
```

Should show:

```
NAME          STATUS    PORTS
wainlux-k6    Up        0.0.0.0:8080->8080/tcp
```

3.3.2.5. Test

Find Pi IP:

```
hostname -I
```

Browse:

```
http://<pi-ip>:8080
```

Test sequence:

1. Click CONNECT
2. Wait for CONNECTED status
3. Click DRAW BOUNDS
4. K6 should trace perimeter
5. Click HOME

3.3.2.6. Logs

```
docker compose logs -f
```

Look for:

```
INFO:app.k3:K3 connected
INFO:werkzeug:127.0.0.1 - - "GET / HTTP/1.1" 200
```

3.3.3. Troubleshooting

3.3.3.1. Build fails: memory

```
free -h
# Check swap
```

```
sudo dphys-swapfile swapoff
sudo nano /etc/dphys-swapfile
# CONF_SWAPSIZE=1024
sudo dphys-swapfile setup
sudo dphys-swapfile swapon
```

3.3.3.2. K6 not found

```
lsusb | grep 10c4
```

Should show:

```
Bus 001 Device 004: ID 10c4:ea60 Silicon Labs CP210x UART Bridge
```

Check serial:

```
ls -la /dev/ttyUSB*
# Should show: /dev/ttyUSB0
```

If missing:

- Check USB cable
- Try different port
- Reboot Pi

3.3.3.3. Container won't start

```
docker compose logs
```

Common issues:

- Port 8080 in use: Change in compose.yaml
- USB permission: Check privileged mode
- Missing image: Rebuild

3.3.3.4. Port in use

Change port in docker-wainlux/compose.yaml:

```
ports:
  - "8081:8080"
```

3.3.4. Auto-start

Create service:

```
sudo nano /etc/systemd/system/wainlux.service
```

```
[Unit]
Description=Wainlux K6 Interface
Requires=docker.service
After=docker.service

[Service]
Type=oneshot
RemainAfterExit=yes
WorkingDirectory=/home/pi/wainlux-pi/docker-wainlux
ExecStart=/usr/bin/docker compose up -d
ExecStop=/usr/bin/docker compose down
User=pi

[Install]
WantedBy=multi-user.target
```

Enable service:

```
sudo systemctl enable wainlux
sudo systemctl start wainlux
```

3.3.5. Maintenance

3.3.5.1. Update

Pull changes:

```
cd ~/wainlux-pi
git pull
```

Rebuild:

```
cd docker-wainlux
docker compose down
docker compose build --no-cache
docker compose up -d
```

3.3.5.2. Uninstall

```
cd ~/wainlux-pi/docker-wainlux
docker compose down
docker rmi docker-wainlux-wainlux
```

```
cd ~
rm -rf wainlux-pi
```

3.4. Application Implementation (PENDING)

3.4.1. Introduction

Pi Zero W controls Wainlux K6 laser via USB serial.

Docker container. Web interface. Binary raster protocol.

3.4.1.1. Features

- Web UI on port 8080
- Test patterns
- Image upload and engraving
- Direct serial protocol
- Docker containerized

3.4.2. Quick start

See Installation and Setup for detailed installation instructions.

3.4.2.1. Prerequisites

1. Pi Zero W with OS
2. Docker installed
3. 1GB swap enabled
4. K6 connected via USB
5. WiFi network

3.4.2.2. Build and run

```
cd docker-wainlux
docker compose build
docker compose up -d
```

3.4.2.3. Access

Open browser: `http://<pi-ip>:8080`

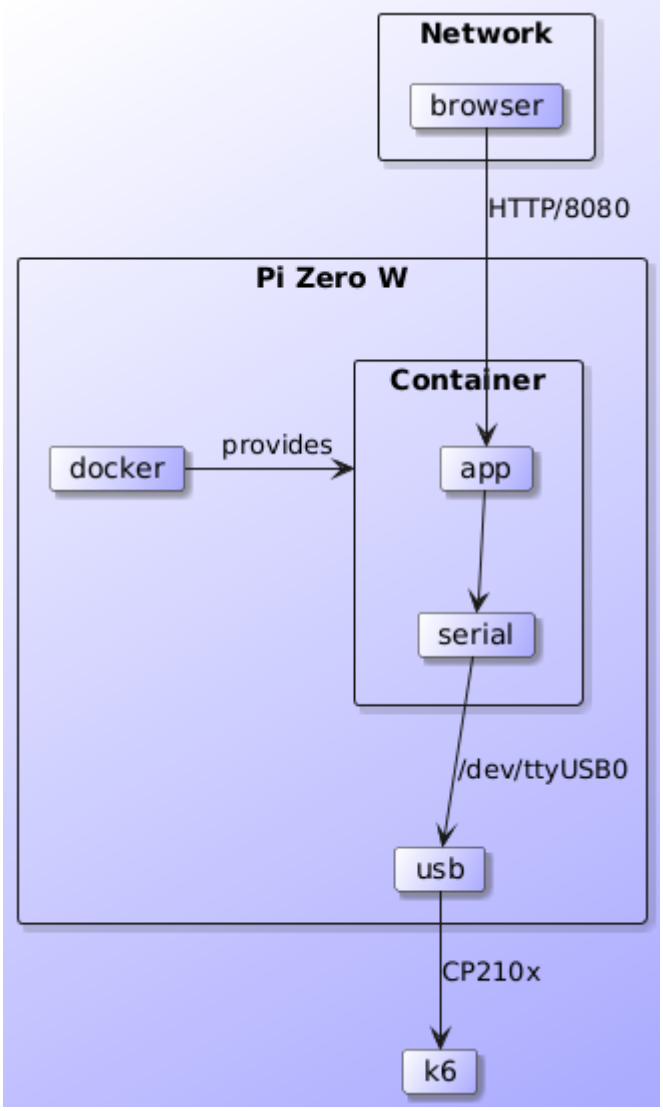
3.4.2.4. First steps

1. Click CONNECT

2. Click DRAW BOUNDS (test pattern)
3. Upload image
4. Click BURN

3.4.3. Architecture

3.4.3.1. System overview



3.4.3.2. Components

3.4.3.2.1. Pi Zero W

- Docker host
- Serial USB device
- WiFi client

3.4.3.2.2. Container

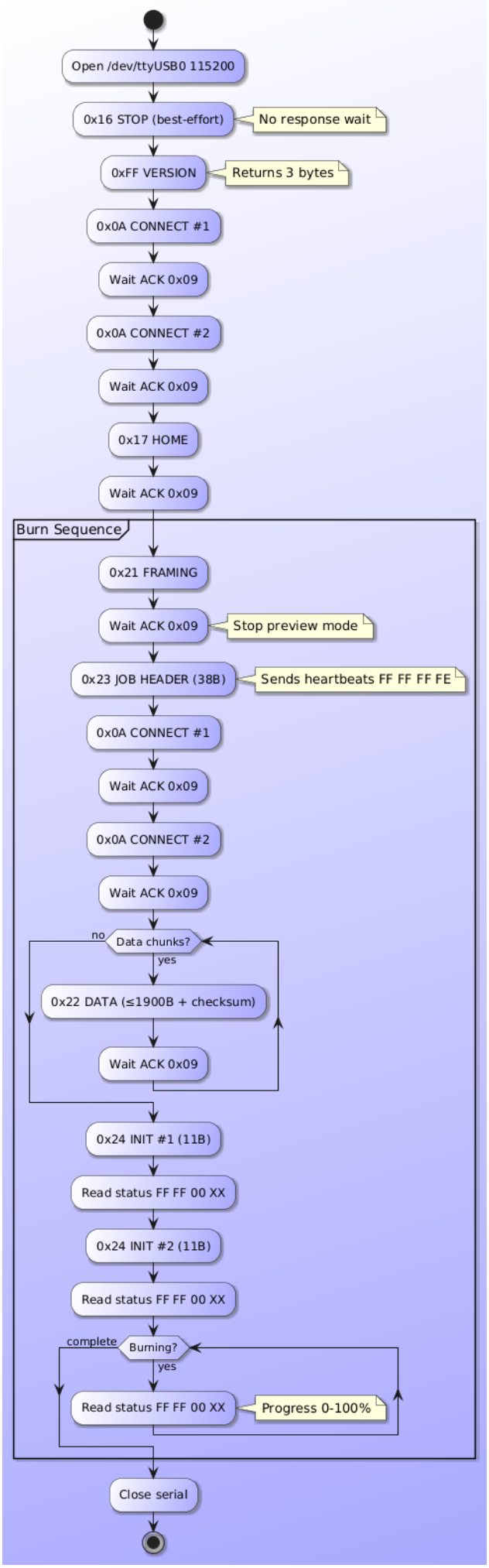
- Flask web app
- K6 protocol driver

- pyserial communication

3.4.3.2.3. Browser

- User interface
- Image upload
- Status display

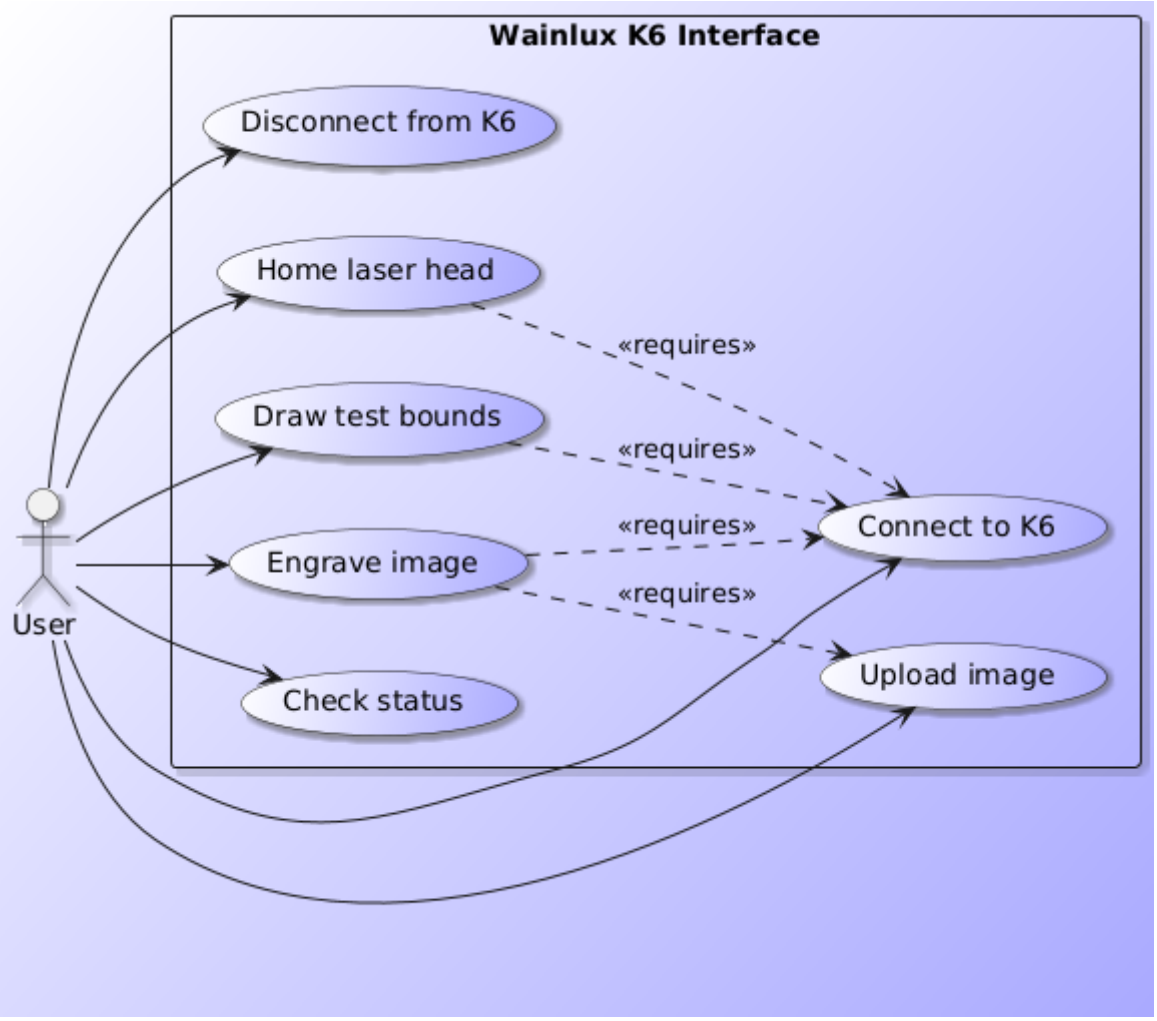
3.4.3.3. Data flow


Open /dev/ttyUSB0 115200
0x16 STOP (best-effort)
No response wait
0xFF VERSION
Returns 3 bytes
0x0A CONNECT #1
Wait ACK 0x09
0x0A CONNECT #2
Wait ACK 0x09
0x17 HOME
Wait ACK 0x09
Burn Sequence
0x21 FRAMING
Wait ACK 0x09
Stop preview mode
0x23 JOB HEADER (38B)
Sends heartbeats FF FF FF FE
0x0A CONNECT #1
Wait ACK 0x09
0x0A CONNECT #2
Wait ACK 0x09
no
Data chunks?
yes
0x22 DATA (≤1900B + checksum)
Wait ACK 0x09
0x24 INIT #1 (11B)
Read status FF FF 00 XX
0x24 INIT #2 (11B)
Read status FF FF 00 XX
complete
Burning?
yes
Read status FF FF 00 XX
Progress 0-100%
Close serial

Serial protocol:

1. Open /dev/ttyUSB0 at 115200 baud
2. Send binary command
3. Wait for ACK byte (9)
4. Repeat or close

3.4.4. Use cases



3.4.4.1. UC1: Connect to K6

Actor: User

Precondition: K6 plugged into Pi via USB

Flow:

1. User opens web interface
2. User clicks CONNECT button
3. System opens /dev/ttyUSB0
4. System confirms connection
5. Status shows CONNECTED

Postcondition: K6 ready for commands

3.4.4.2. UC2: Disconnect from K6

Actor: User

Precondition: K6 connected

Flow:

1. User clicks DISCONNECT button
2. System closes serial port
3. Status shows DISCONNECTED

Postcondition: K6 released

3.4.4.3. UC3: Home laser head

Actor: User

Precondition: K6 connected

Flow:

1. User clicks HOME button
2. System sends home command
3. K6 moves to origin
4. System waits for ACK

Postcondition: Laser at home position

3.4.4.4. UC4: Draw test bounds

Actor: User

Precondition: K6 connected

Flow:

1. User clicks DRAW BOUNDS button
2. System generates boundary image
3. System engraves perimeter
4. Laser traces rectangle

Postcondition: Test pattern visible

3.4.4.5. UC5: Upload image

Actor: User

Precondition: None

Flow:

1. User clicks file input
2. User selects image file
3. Browser validates format
4. File ready for engraving

Postcondition: Image loaded in browser

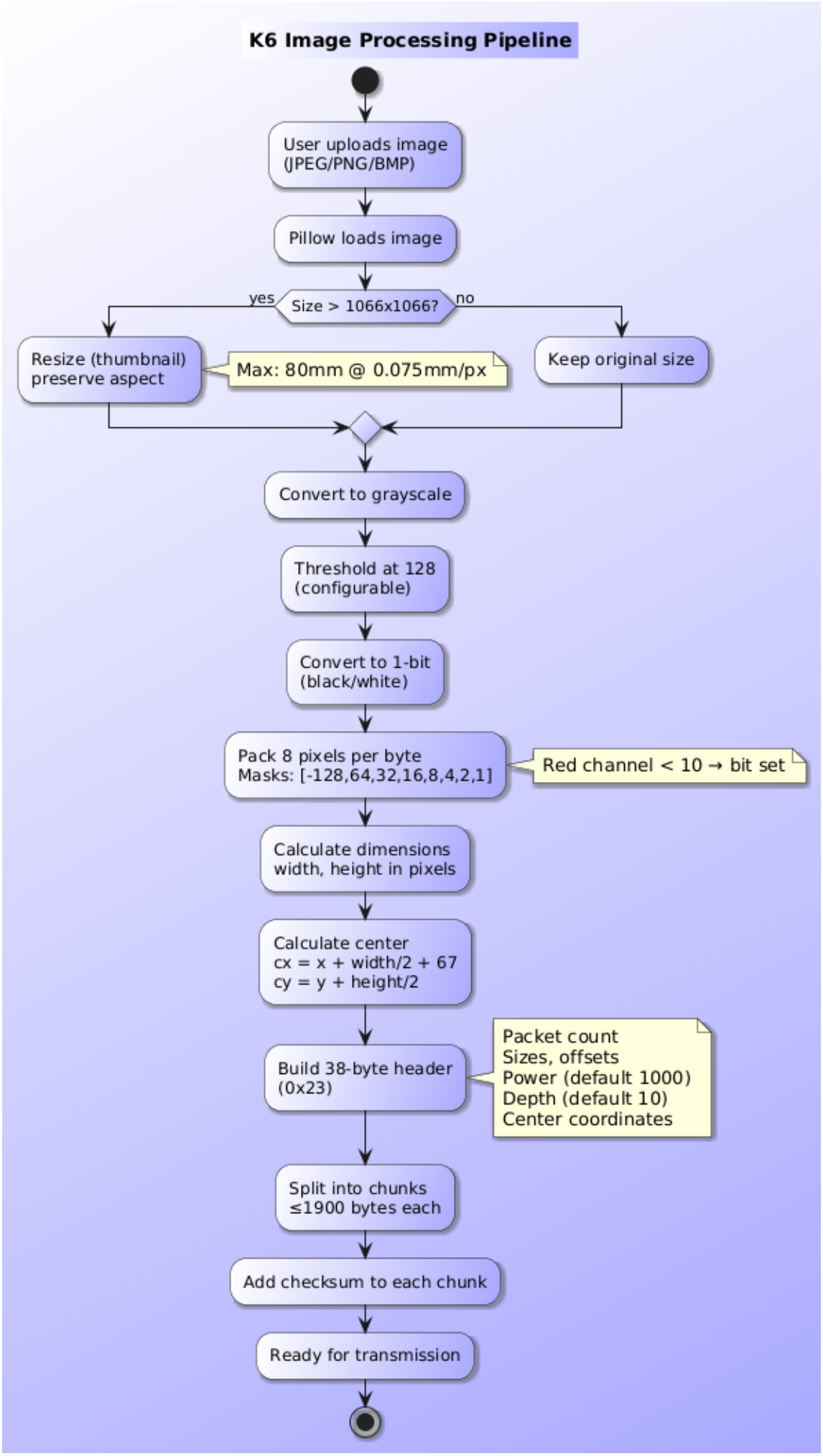
3.4.4.6. UC6: Engrave image

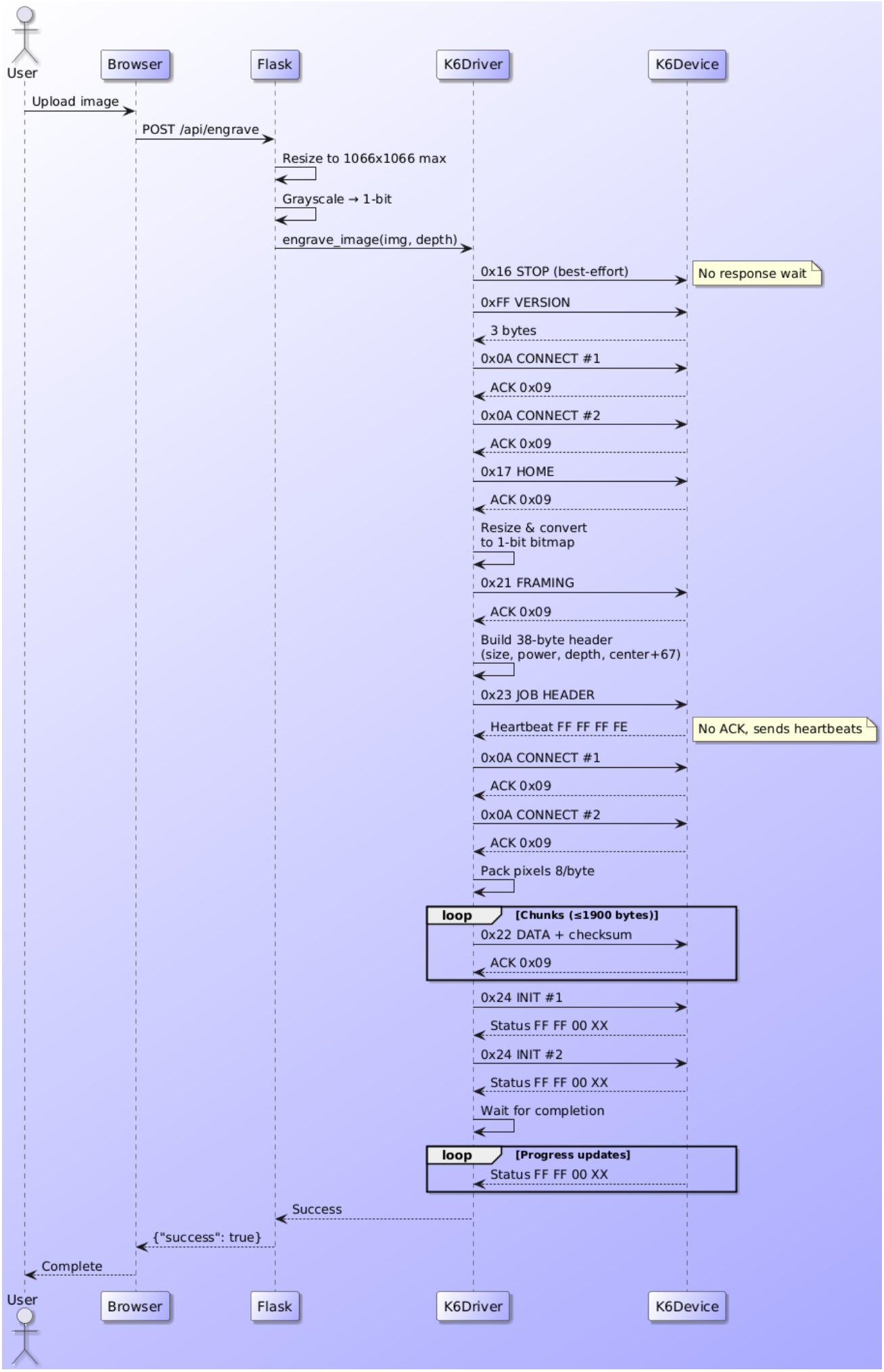
Actor: User

Precondition: K6 connected, image uploaded

Flow:

1. User clicks BURN button
2. System processes image
3. System converts to 1-bit
4. System packs pixels
5. System sends chunked data
6. K6 engraves image


K6 Image Processing Pipeline
User uploads image (JPEG/PNG/BMP)
Pillow loads image
Size > 1066x1066?
yes
no
Resize (thumbnail) preserve aspect
Max: 80mm @ 0.075mm/px
Keep original size
Convert to grayscale
Threshold at 128 (configurable)
Convert to 1-bit (black/white)
Pack 8 pixels per byte
Masks: [-128,64,32,16,8,4,2,1]
Red channel < 10 → bit set
Calculate dimensions width, height in pixels
Calculate center
cx = x + width/2 + 67
cy = y + height/2
Build 38-byte header (0x23)
Packet count
Sizes, offsets
Power (default 1000)
Depth (default 10)
Center coordinates
Split into chunks ≤1900 bytes each
Add checksum to each chunk
Ready for transmission


User
Browser
Flask
K6Driver
K6Device
Upload image
POST /api/engrave
Resize to 1066x1066 max
Grayscale → 1-bit
engrave_image(img, depth)
0x16 STOP (best-effort)
No response wait
0xFF VERSION
3 bytes
0x0A CONNECT #1
ACK 0x09
0x0A CONNECT #2
ACK 0x09
0x17 HOME
ACK 0x09
Resize & convert
to 1-bit bitmap
0x21 FRAMING
ACK 0x09
Build 38-byte header
(size, power, depth, center+67)
0x23 JOB HEADER
Heartbeat FF FF FF FE
No ACK, sends heartbeats
0x0A CONNECT #1
ACK 0x09
0x0A CONNECT #2
ACK 0x09
Pack pixels 8/byte
loop
[Chunks (≤1900 bytes)]
0x22 DATA + checksum
ACK 0x09
0x24 INIT #1
Status FF FF 00 XX
0x24 INIT #2
Status FF FF 00 XX
Wait for completion
loop
[Progress updates]
Status FF FF 00 XX
Success
{"success": true}
Complete
User
Browser
Flask
K6Driver
K6Device

Postcondition: Image engraved on material

3.4.4.7. UC7: Check status

Actor: User

Precondition: None

Flow:

1. User opens interface
2. System displays connection state
3. System shows max dimensions

Postcondition: User informed

3.5. K6 Protocol Reference (Complete)

This file consolidates all K6 protocol documentation: runtime commands, job header parameters, and firmware update protocol.

3.5.1. Overview & Safety

3.5.1.1. Scope

This documents the **observed** K6 protocol from the vendor macOS Java app. It is not final. It is the current best-known set of commands for raster mode and partial vector mode.

3.5.1.2. Safety Warning

Laser safety first. Do not run tests unattended. Use eye protection and a fire-safe surface.

Recent tests caused runaway motion:

- Single-line raster test drove into X limit for >10s, moved Y, returned to top-left and kept pushing.
- Init-only test (`0x24` twice) moved X+ then X- then X+ and kept pushing.
- After power-cycle, the device resumed motion immediately.

Treat this as a runaway state. Be ready to power off immediately and avoid repeat until the command sequence is verified.

3.5.2. Vendor Specs & Hardware

3.5.2.1. Specification

Specification of K6 engraving&cutting machine

- Product Name: Wainlux K6 Mini Laser Engraving Machine

- Brand: Wainlux
- Rated power: 3W
- Product weight: 0.96kg
- Product size: 167x167x165mm
- Engraving area: 80x80mm (vendor spec)
- Supported OS: Windows/MAC/IOS/Android
- Support format: JPEG/JPG/PNG/BMP
- Vendor page: https://www.wainlux.com/products/wainlux-k6-mini-laser-engraving-machine?variant=40305160192086

3.5.2.2. Hardware Details

- Engraving area: 80x80mm (vendor spec)
- Observed Y-axis: 76mm (4mm shortfall, needs calibration investigation)
- USB: CP2102 serial (10c4:ea60)
- Protocol: Custom (NOT GRBL)
- Baud: 115200

3.5.2.3. Coordinates & Units

- Origin: Top-left
- Units: Pixels
- X: Left to right (0-1066 max)
- Y: Top to bottom (0-1066 max)

3.5.2.4. Resolution and Limits

- **Hardware resolution: 0.05 mm/pixel (FIXED BY FIRMWARE)**
- Work area: 80mm ÷ 0.05 = 1600 pixels
- **Max raster: 1600x1600 pixels** (observed and verified.. and now thrown into doubt after measuring bounds shortfall on Y axis)
- Alternative resolutions documented (0.0625, 0.075, 0.08, 0.096 mm/pixel) **do not work correctly**
- K3 limits (different): 1600x1520px (for reference only - NOT K6.... or maybe are!!)
- Depth: 1-255 (laser on time, default 10)
- Power: 0-1000 (UI value x 10, default 1000)
- Repeat count: 1-10 (default 1)

Firmware uses fixed 0.05 mm/px resolution regardless of image size.

Test results (26-Jan-2026):

- 0.05 mm/px, 10mm target (200px) → 10mm actual ✔
- 0.08 mm/px, 10mm target (125px) → 6mm actual (125px × 0.05 = 6.25mm)

The stepper motor step size is hardware-defined. Sending different pixel counts does not change physical resolution - the firmware always interprets pixels as 0.05mm spacing.

For accurate dimensions: always use 0.05 mm/px (1600×1600 = 80×80mm)

3.5.2.5. ACK Protocol

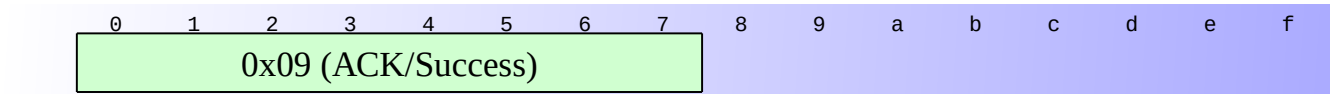


Figure 1. K6 ACK Response (0x09) (`k6-prot-rx-09-ack.edn`)

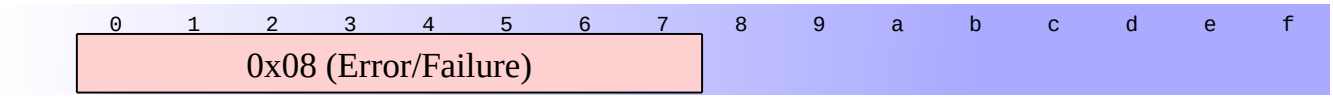


Figure 2. K6 Error Response (0x08) (`k6-prot-rx-08-error.edn`)

- ACK byte: `0x09` (success)
- Error byte: `0x08` (failure)
- Most commands expect ACK (timeout 1-3s)
- Exceptions: Job header (0x23) does not wait for ACK
- Status frames: `FF FF 00 XX` for progress reporting

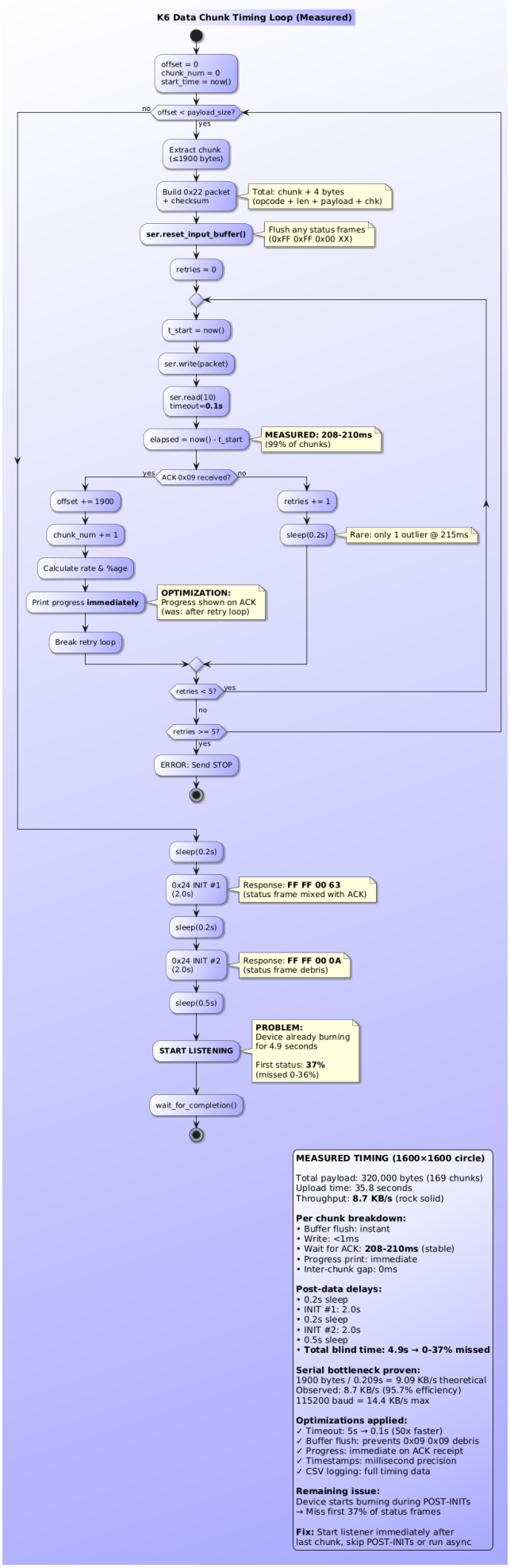


Figure 3. Data chunk timing sequence

Each data chunk (opcode `0x23`) requires ACK `0x09` before sending next chunk. 100ms pause after motion commands.

3.5.2.6. USB Serial Setup

pyserial setup

```
import serial

ser = serial.Serial(
    port='/dev/ttyUSB0',
    baudrate=115200,
    timeout=2
)
```

- Device: /dev/ttyUSB0
- Baud: 115200
- Binary protocol (not text-based)
- Checksum: see Checksum Algorithm

3.5.2.7. Vector vs Raster

The first "vector" test of a small circle went well **AND** it's clear that the circle was burnt line by line (raster) and **not** as a vector (continuous line).

The first raster tests also went well and also went line by line.

Vector mode on the K6 does not draw continuous lines. It burns line-by-line. Same as raster. You send point coordinates. The device converts them to a bitmap. Then it burns the bitmap. Horizontally. Line by line. The only difference: data format. Vector sends x,y pairs. Raster sends pixels. Both burn the same way. Why use vector? Fewer bytes for simple shapes. That's all. Vector on K6 = vector-defined raster.

3.5.2.7.1. Raster or Vector. Not both.

Each JOB_HEADER command specifies either raster or vector parameters. Not both. The header sets the mode. One mode per job. That's the limit.

3.5.3. Runtime Commands

3.5.3.1. Status

- Protocol is partially confirmed by live tests on the Pi.
- Raster mode commands are the most complete.
- Vector mode is partially understood (point list in job data), not yet verified.

3.5.3.2. Live Verification (Pi)

Test run: `test_mvp_mac_proto.py` on `pi-hostname` (Raspbian armv6).

Confirmed:

- `0xFF` version returns 3 bytes (example: `04 01 06`).
- `0x0A` connect ACKs with `0x09` (twice).
- `0x17` home ACKs with `0x09`.
- `0x21` framing ACKs with `0x09`.
- `0x24` init/status returns `ff ff 00 00`.

Not confirmed:

- `0x22` job data timed out on a single long chunk in this run. Likely needs chunking or pacing (observed: 1900-byte chunks with retry on timeout).

KISS single-line test (1600x1, no padding):

- Responses were `ff ff ff fe` for version/connect/home/framing in this run.
- `0x22` data timed out.
- `0x24` init timed out once, then returned `ff ff ff fe`.

Interpretation: device likely in a bad state. Power-cycle is recommended before retry.

3.5.3.3. Command Summary

Opcode	Name	Cmd Length	ACK/Response	Discovery	Notes
`0xFF`	Version	4	3-byte reply	Observed	Request: `ff 00 04 00`, Response: major, minor, patch
`0x0A`	Connect	4	`0x09`	Observed	Sent twice before job data
`0x06`	Crosshair ON	4	`0x09`	Ghidra via LLM	Enable positioning laser (no observable effect)
`0x07`	Crosshair OFF	4	`0x09`	Ghidra via LLM	Disable positioning laser (no observable effect)
`0x16`	Stop/Cancel	4	`0x09`	Ghidra via LLM	Cancel current operation (unreliable)
`0x17`	Home	4	`0x09`	Observed	Move to origin (0,0)
`0x20`	Set Bounds	11	`0x09`	Ghidra via LLM	Enable preview mode - trace bounds outline

Opcode	Name	Cmd Length	ACK/Response	Discovery	Notes
0x21	Framing/box	4	0x09	Observed	Disable preview mode - stop bounds tracing
0x23	Job header	38	none waited	Observed	Starts a job
0x22	Job data	variable	0x09	Observed	Chunked payload + checksum
0x24	Init/status	11	status frame	Observed	Sent after job data
0x25	Set Speed/Power	11	0x09	Ghidra via LLM	Configure speed and power
0x28	Set Focus/Angle	11	0x09	Ghidra via LLM	Set focus and angle params

3.5.3.4. Response Frames

3.5.3.4.1. ACK / Error Byte

- 0x09 = OK
- 0x08 = error

3.5.3.4.2. Heartbeat Frame (Processing)

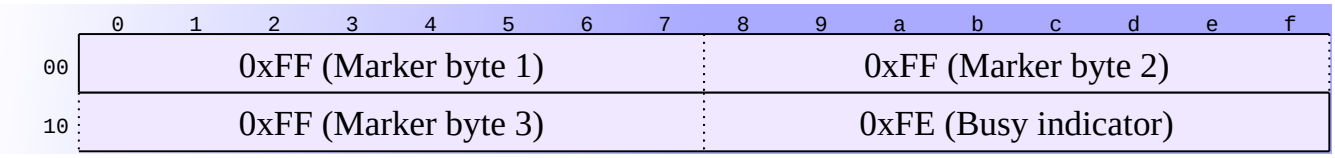


Figure 4. K6 Heartbeat Frame (k6-prot-rx-heartbeat.edn)

FF FF FF FE - sent every ~4 seconds while device processes a job header or large command.

- Observed after JOB_HEADER (0x23) - device sends 7 heartbeats over ~28s
- No ACK follows heartbeat frames
- Indicates device is busy calculating/preparing job

3.5.3.4.3. Status Frame (Progress)

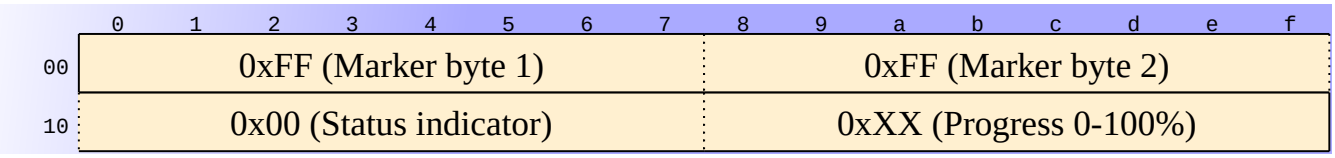


Figure 5. K6 Status Frame (k6-prot-rx-status.edn)

FF FF 00 XX where XX is a progress percentage (0-100).

- Sent during burn operation

3.5.3.5. Packet Formats

3.5.3.5.1. Connect (0x0A)

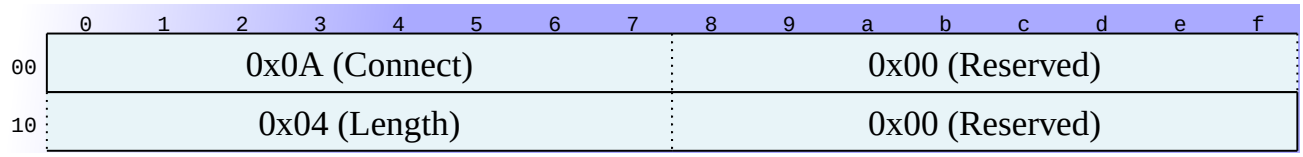


Figure 6. K6 Connect Command (0x0A) (`k6-prot-tx-0a-connect.edn`)

Sent twice with 500 ms delay between sends.

After JOB_HEADER: Two CONNECTs serve different purposes: - **CONNECT #1**: Flushes heartbeat buffer (gets 2-3 heartbeats + ACK) - **CONNECT #2**: Clean communication (gets immediate ACK)

The JOB_HEADER triggers continuous heartbeats that stream in background. First CONNECT drains these, second gets clean channel.

State Management: - **Not required** for movement commands (BOUNDS, JOG, HOME work without CONNECT) - **Not enforced** by firmware - device accepts commands when "disconnected" - **Recommended** at session start to establish sane device state - **Not tracked** by device - connection is client-side concept only

3.5.3.5.2. Version (0xFF)

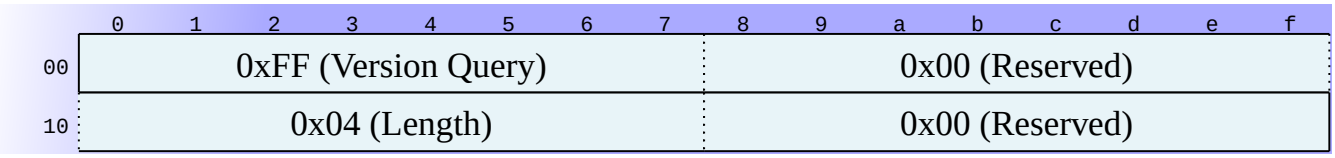


Figure 7. K6 Version Query (0xFF)

Expects a 3-byte reply (example: `04 01 06`).

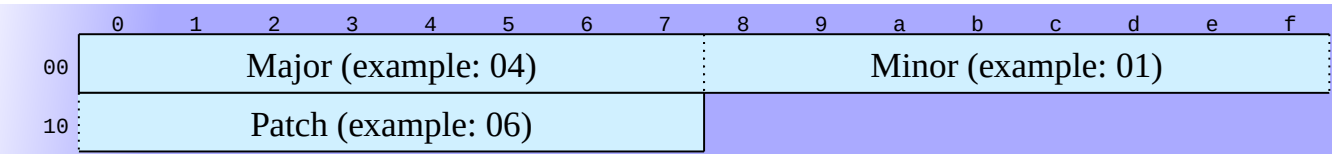


Figure 8. K6 Version Response (`k6-prot-rx-version.edn`)

3.5.3.5.3. Job Header (0x23)

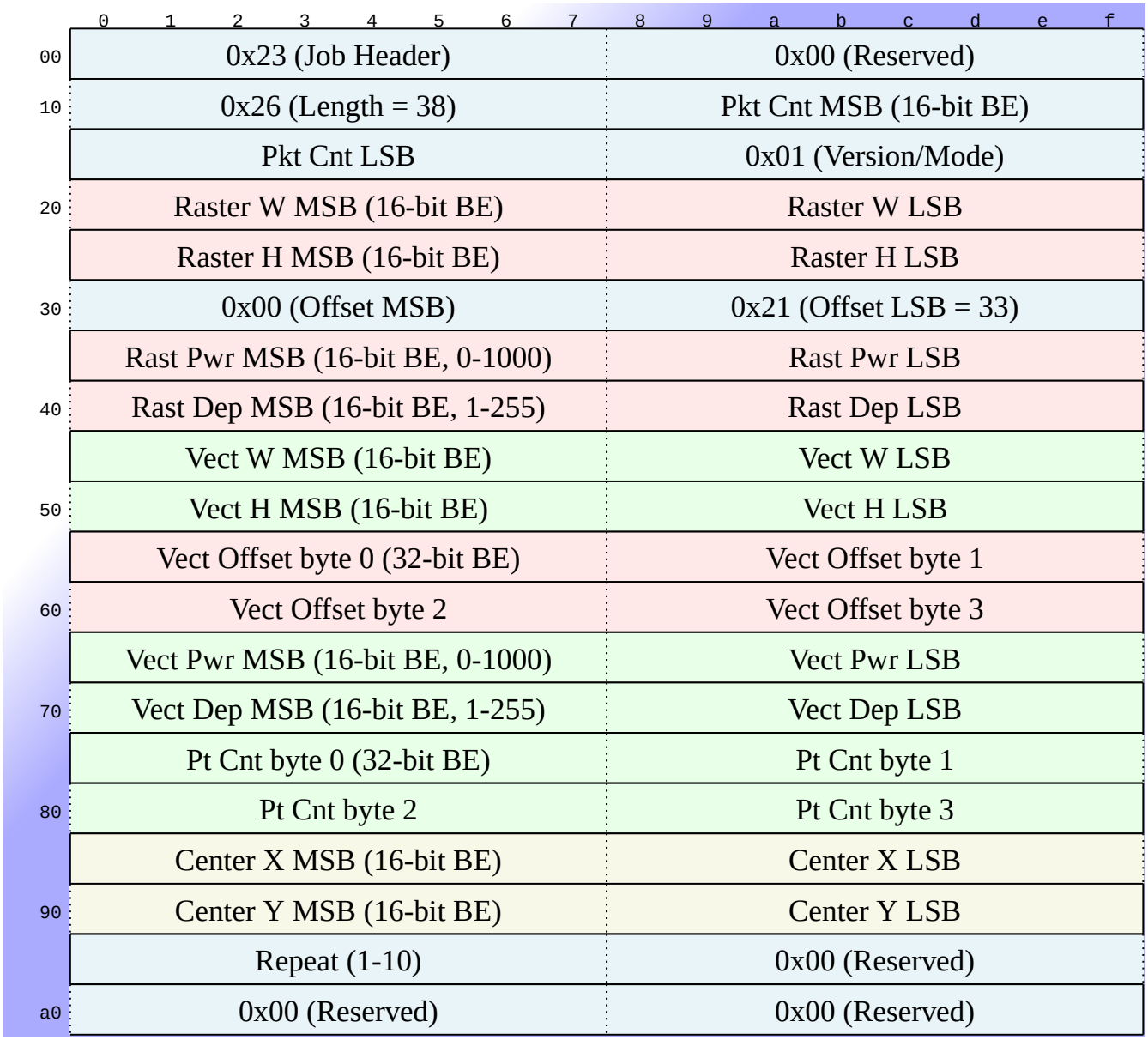


Figure 9. K6 Job Header (0x23) (`k6-prot-tx-23-job-header.edn`)

Total length: 38 bytes.

Byte layout observed:

- `byte[0] = 0x23`
- `byte[1] = 0x00`
- `byte[2] = 0x26` (38)
- Remaining bytes are parameters (sizes, offsets, settings)
- `byte[37] = 0x00`

Notes:

- **No ACK sent** - device sends heartbeat frames instead
- **Heartbeat frames** (`FF FF FF FE`) sent every ~4 seconds continuously
- Device does NOT stop sending heartbeats - they continue indefinitely

- First heartbeat arrives ~4.7s after sending JOB_HEADER
- Proceed to next command after receiving first heartbeat
- Parameters include width, height, position, and speed/power settings.
- All multi-byte fields are **big-endian** (MSB first).
- Raster width/height are raw pixel dimensions (no scaling applied).
- Raster depth default = `10` (not `1`).
- Center offsets use +67 formula (see Job Header Parameters).
- Work area observed: 80x80mm, resolution 0.075 mm/pixel ⇒ max ≈ `1067 x 1067` px.

See Job Header Parameters for complete byte-level mapping.

3.5.3.5.4. Job Data (0x22)

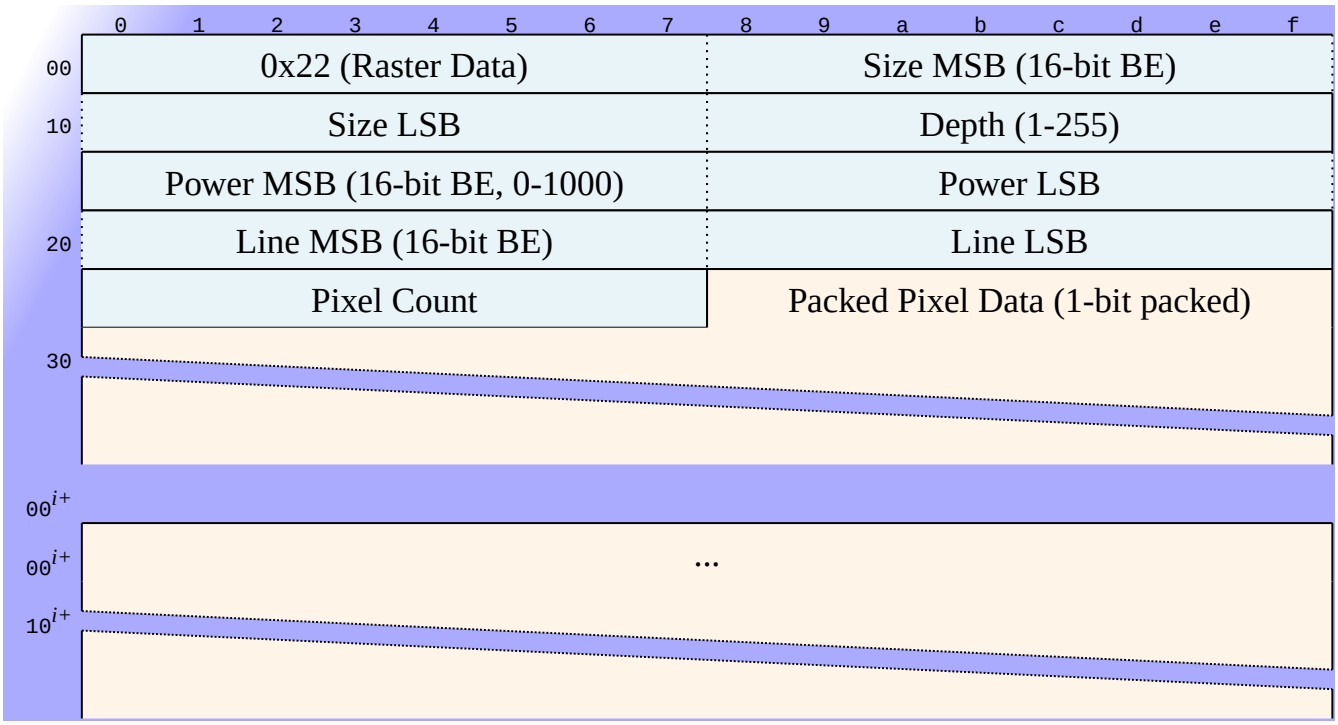


Figure 10. K6 Job Data (0x22) (`k6-prot-tx-22-data.edn`)

Chunk size: 1900 bytes of payload.

Packet layout (per chunk):

- `byte[0] = 0x22`
- `byte[1] = length >> 8`
- `byte[2] = length`
- `byte[3..N-2] = payload`
- `byte[N-1] = checksum` (see Checksum Algorithm)

Observed retry behavior: chunk is retransmitted if ACK not received.

Device rejects raster data with excessive blank lines (error 0x08).

Test results (28-Jan-2026):

- Sending 1600x1600 image with 1400+ blank lines at top → device returns error 0x08
- Sending only non-blank region (e.g., 1600x210 for bottom content) → success

Workaround: Crop images to only the region with content before sending. Use center_y coordinate in JOB_HEADER to position at bottom/top as needed.

- For bottom positioning: `center_y = 1600 - (height / 2)`
- For center positioning: `center_y = 800` (middle of 1600px work area)
- For top positioning: `center_y = height / 2`

This appears to be a firmware limitation or optimization - the device may reject jobs with excessive preprocessing requirements.

Live test note:

- A single long chunk timed out on Pi.
- Use 1900-byte chunks with retry on timeout.

3.5.3.5.5. Init/Status (0x24)

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	0x24 (Init/Status)								0x00 (Reserved)							
10	0x0B (Length = 11)								0x00 (Reserved)							
	0x00 (Reserved)								0x00 (Reserved)							
20	0x00 (Reserved)								0x00 (Reserved)							
	0x00 (Reserved)								0x00 (Reserved)							
30	0x00 (Reserved)								0x00 (Reserved)							

Figure 11. K6 Init/Status Command (0x24) (`k6-prot-tx-24-init.edn`)

11-byte packet, sent twice after job data:

- `byte[0] = 0x24`
- `byte[1] = 0x00`
- `byte[2] = 0x0B`
- Remaining bytes are zero

This triggers or accompanies the `ff ff 00 xx` status frames.

3.5.3.5.6. Crosshair (0x06, 0x07)

Toggle positioning laser/LED for alignment:

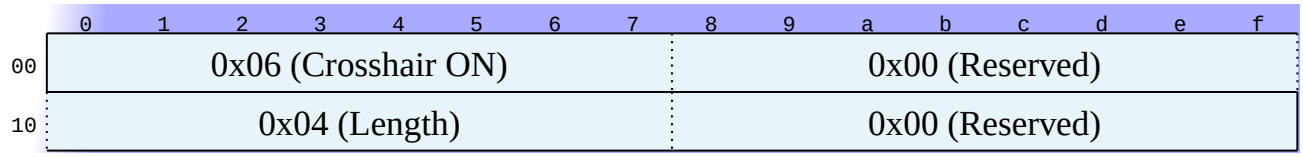


Figure 12. K6 Crosshair ON (0x06) (k6-prot-tx-06-crosshair-on.edn)

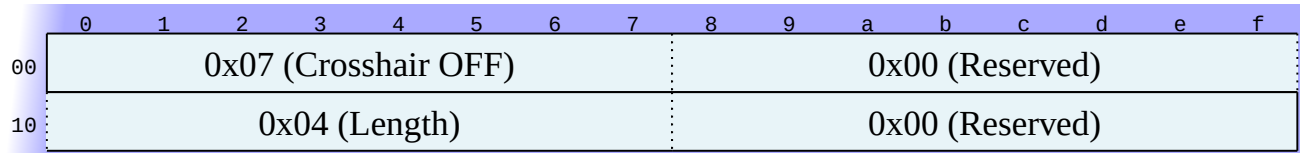


Figure 13. K6 Crosshair OFF (0x07) (k6-prot-tx-07-crosshair-off.edn)

- **Purpose:** Enable/disable positioning laser for alignment before burning
- **Length:** 4 bytes each
- **ACK:** Expected (0x09, timeout 2s)
- **Usage:** Send 0x06 to enable, 0x07 to disable
- **Observed Behavior:** No visible effect observed in testing - purpose unclear
- **Status:** Available for testing but not used in production sequences
- **Discovery:** Ghidra analysis via LLM via MCP

3.5.3.5.7. Stop/Cancel (0x16)

Stop current operation immediately:

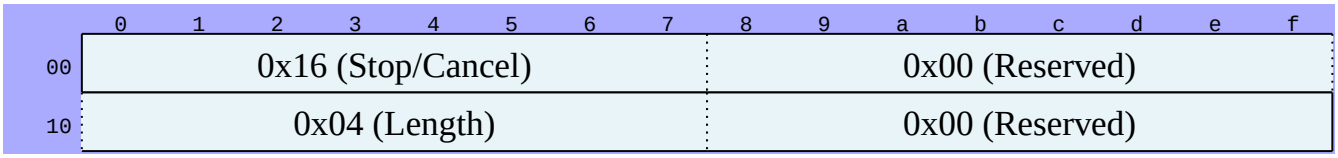


Figure 14. K6 Stop/Cancel Command (0x16) (k6-prot-tx-16-stop.edn)

- **Purpose:** Cancel active engraving/motion
- **Length:** 4 bytes
- **ACK:** Expected (0x09, timeout 2s)
- **Usage:** Critical safety feature - should be easily accessible
- **Observed Behavior:** Unreliable for state clearing - device may not fully reset
- **Note:** Use explicit mode-exit commands (0x21 FRAMING) instead of relying on STOP for state management
- **Discovery:** Ghidra analysis via LLM via MCP

3.5.3.5.8. Home (0x17)

Return to home position:

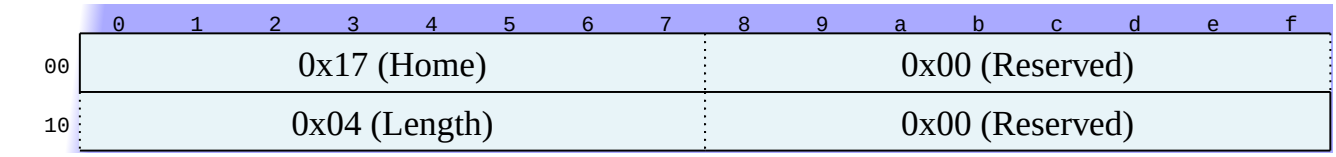


Figure 15. K6 Home Command (0x17) (k6-prot-tx-17-home.edn)

- **Purpose:** Move to origin/home position
- **Length:** 4 bytes
- **ACK:** Expected (0x09, timeout 10s)
- **Usage:** Initialize position before operations

3.5.3.5.9. Framing (0x21)

Draw boundary frame:

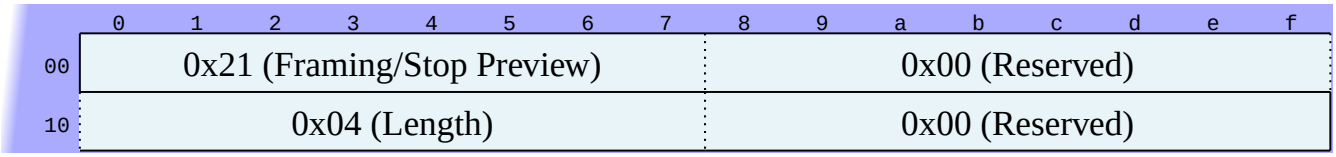


Figure 16. K6 Framing Command (0x21) (k6-prot-tx-21-framing.edn)

- **Purpose:** Stop preview mode / draw boundary box for work area preview
- **Length:** 4 bytes
- **ACK:** Expected (0x09, timeout 2s)
- **Behavior:** When sent after 0x20 BOUNDS command, stops continuous preview tracing and returns laser head to center position specified in BOUNDS command (NOT origin)
- **Usage:**
  - Sent before burn sequence to disable bounds checking/preview mode
  - To return to origin (0,0) after stopping preview, must send 0x17 HOME after 0x21 FRAMING
  - Called "preview OFF" - counterpart to 0x20 BOUNDS which is "preview ON"
- **Note:** 0x20 starts preview loop, 0x21 stops it

3.5.3.5.10. Set Bounding Box (0x20)

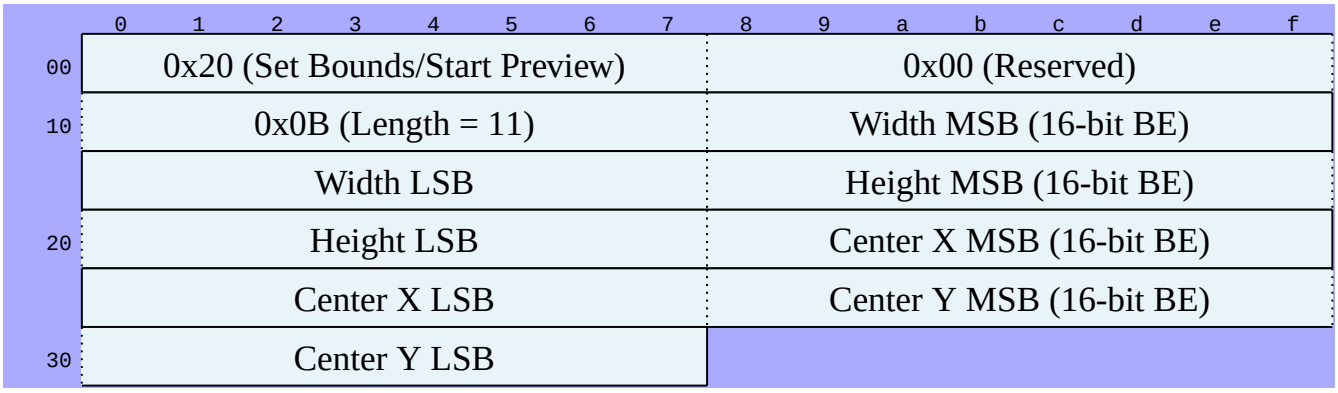


Figure 17. K6 Set Bounding Box (0x20) (k6-prot-tx-20-bounds.edn)

Configure bounding box for framing area:

- **Purpose:** Set bounding box for selection/framing area, starts preview mode
- **Length:** 11 bytes
- **ACK:** Expected (0x09, timeout 1s)
- **Behavior:** Enables preview mode - laser traces bounds outline (called "framing ON" or "preview ON")
- **Parameters:**
  - Bytes 3-4: width (16-bit BE)
  - Bytes 5-6: height (16-bit BE)
  - Bytes 7-8: center_x (16-bit BE)
  - Bytes 9-10: center_y (16-bit BE)
- **Note:** Uses same +67 centering formula as job header (see Job Header Parameters)
- **Discovery:** Ghidra analysis via LLM via MCP

3.5.3.5.11. Set Speed/Power (0x25)

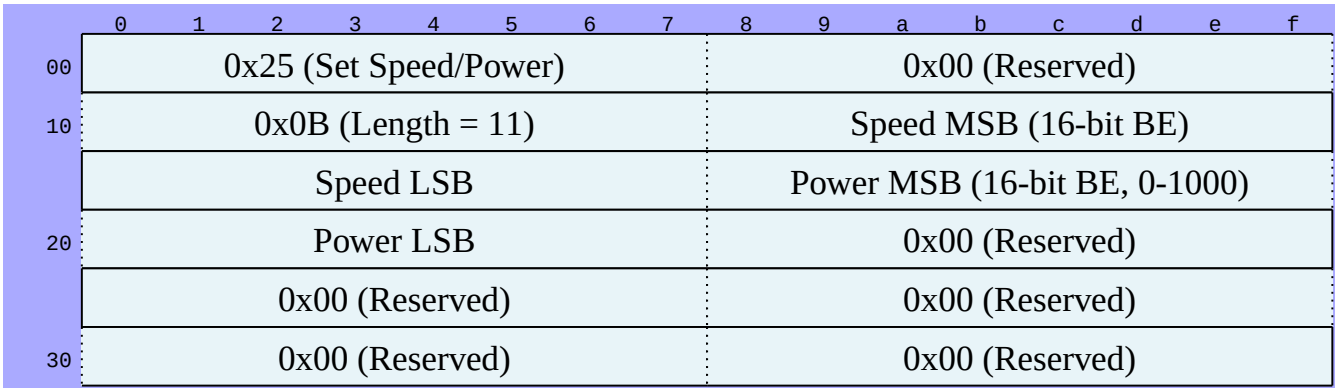


Figure 18. K6 Set Speed/Power (0x25) (`k6-prot-tx-25-speed-power.edn`)

Configure speed and power settings:

- **Purpose:** Set speed and power (separate from job header)
- **Length:** 11 bytes
- **ACK:** Expected (0x09, timeout 2s)
- **Parameters:**
  - Bytes 3-4: speed (16-bit BE)
  - Bytes 5-6: power (16-bit BE)
  - Bytes 7-10: reserved (zeros)
- **Note:** Precedence vs job header values needs hardware testing
- **Discovery:** Ghidra analysis via LLM via MCP

3.5.3.5.12. Set Focus/Angle (0x28)

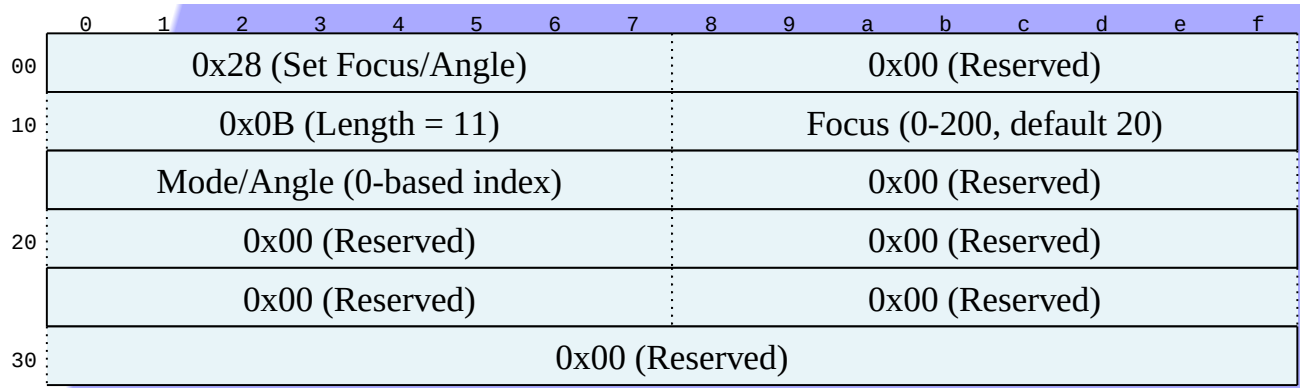


Figure 19. K6 Set Focus/Angle (0x28) (`k6-prot-tx-28-focus.edn`)

Configure focus/angle parameters:

- **Purpose:** Set focus and angle parameters
- **Length:** 11 bytes
- **ACK:** Expected (0x09, timeout 2s)
- **Parameters:**
  - Byte 3: focus parameter (default 20, range 0-200, typically UI_value x 2)
  - Byte 4: mode/angle index (0-based, purpose unclear)
  - Bytes 5-10: reserved (zeros)
- **Note:** "Weak light power" - likely crosshair/positioning laser intensity
- **Note:** Actual effect needs hardware testing
- **Discovery:** Ghidra analysis via LLM via MCP

3.5.3.6. Raster Bit Packing

Observed pixel packing:

- Pack 8 pixels per byte with masks: `[-128, 64, 32, 16, 8, 4, 2, 1]`.
- Use the red channel as threshold.
- If $R < 10$, set the bit.

3.5.3.7. Vector Mode (Partial)

The job payload can include point data after the raster payload:

- Points are appended as pairs of 16-bit values (x, y).
- Each point contributes 4 bytes: `x_lo, x_hi, y_lo, y_hi`.
- Observed in packet structure analysis.

Vector mode is not confirmed on hardware yet.

3.5.3.8. Opcode 0x1c

- Seems to be limited to K3. Not seen on wire.

3.5.3.9. Verification Plan (Pi)

Run only with safety precautions.

- Use `test_mvp_mac_proto.py` for the known-good opcode sequence.
- Verify ACK bytes (`0x09`) for:
  - `0x0A` connect
  - `0x17` home
  - `0x21` framing
  - `0x22` job data chunks
  - `0x06/0x07` crosshair toggle
  - `0x16` stop/cancel
  - `0x20` set bounds
  - `0x25` set speed/power
  - `0x28` set focus/angle
- Confirm version read returns 3 bytes for `0xFF`.
- Confirm status frames `ff ff 00 xx` after `0x24`.
- For vector mode, build a tiny payload with only point data and observe movement.
- Test framing (0x21) before actual burn to verify bounds
- Test stop (0x16) during operation for safety validation
- Test speed/power (0x25) precedence vs job header values
- Test focus/angle (0x28) to understand effect on positioning laser (if any)

3.5.4. Job Header Parameters

This section documents the complete 38-byte job header (opcode 0x23) parameter mapping.

3.5.4.1. Parameter Structure

Header parameters observed:

```
Packet count:        (33 + raster_bytes + vector_bytes) / 4094 + 1
Version/mode:        1
Raster width:        pixel width
Raster height:       pixel height
Raster offset:       33 (constant)
Raster power:        0-1000 (default 1000)
Raster depth:        1-255 (default 10)
Vector width:        vector bounding box width
```

```
Vector height:       vector bounding box height
Vector offset:       33 + raster_size
Vector power:        0-1000 (default 1000)
Vector depth:        1-255 (default 10)
Vector point count:  number of points
Raster center X:     x + width/2 + 67
Raster center Y:     y + height/2
Repeat count:        1-10 (default 1)
Vector center X:     x + width/2 + 67
Vector center Y:     y + height/2
```

Centering Formula:

- Raster and vector both use: `center_x = x + width/2 + 67`, `center_y = y + height/2`
- The +67 offset is consistent for all centering operations

3.5.4.2. Header Byte Layout

Byte(s)	Field	Description
[0-2]	Header	0x23, 0x00, 0x26 (opcode, zero, length 38)
[3-4]	Packet count	16-bit big-endian
[5]	Version/mode	Always 1
[6-7]	Raster width	Pixel width, 16-bit BE
[8-9]	Raster height	Pixel height, 16-bit BE
[10-11]	Raster offset	Data offset, 16-bit BE (always 33)
[12-13]	Raster power	0-1000, 16-bit BE (default 1000)
[14-15]	Raster depth	1-255, 16-bit BE (default 10)
[16-17]	Vector width	Bounds width, 16-bit BE
[18-19]	Vector height	Bounds height, 16-bit BE
[20-23]	Vector offset	Data offset, 32-bit BE = 33 + raster_size
[24-25]	Vector power	0-1000, 16-bit BE (default 1000)
[26-27]	Vector depth	1-255, 16-bit BE (default 10)
[28-31]	Vector points	Point count, 32-bit BE
[32-33]	Raster center X	16-bit BE = x + width/2 + 67
[34-35]	Raster center Y	16-bit BE = y + height/2
[36]	Repeat count	1-10 (default 1)
[37]	Reserved	Always 0

Note: Bytes 32-35 are raster center coordinates. Vector centers are NOT in the header - they appear to be used for calculations but not transmitted.

3.5.4.3. Default Values Observed

- Raster power: default 1000 (range 0-1000, observed via UI: value x 10)
- Raster depth: default 10 (range 1-255)
- Vector power: default 1000 (range 0-1000, observed via UI: value x 10)
- Vector depth: default 10 (range 1-255)
- Contrast/threshold: default 50 (range 0-100)
- Fill density: default 5 (range 0-10)
- Crosshair power: default 20 (range 0-200, observed via UI: value x 2)

3.5.4.4. Work Area and Resolution

Resolution options observed (mm/pixel):

- 0.05, 0.0625, **0.075** (default), 0.08, 0.096, 0.064

Default: 0.075 mm/pixel

- 80mm ÷ 0.075 = 1066.67 pixels (~1067)
- Max safe raster: 1066x1066 pixels (observed limit)

```
# Raster image center (bytes 32-35 in header)
raster_center_x = raster_bbox.x + (raster_bbox.width // 2) + 67
raster_center_y = raster_bbox.y + (raster_bbox.height // 2)

# Vector graphics center (used in calculations, NOT in header)
vector_center_x = vector_bbox.x + (vector_bbox.width // 2) + 67
vector_center_y = vector_bbox.y + (vector_bbox.height // 2)
```

3.5.4.5. Packet Count Calculation

```
packet_count = ((33 + raster_bytes + vector_bytes) // 4094) + 1
```

Where:

- 33 = header size
- `raster_bytes` = raster data size (width x height in bytes for 1-bit packed)
- `vector_bytes` = vector data size (4 bytes per point)
- 4094 = max chunk size

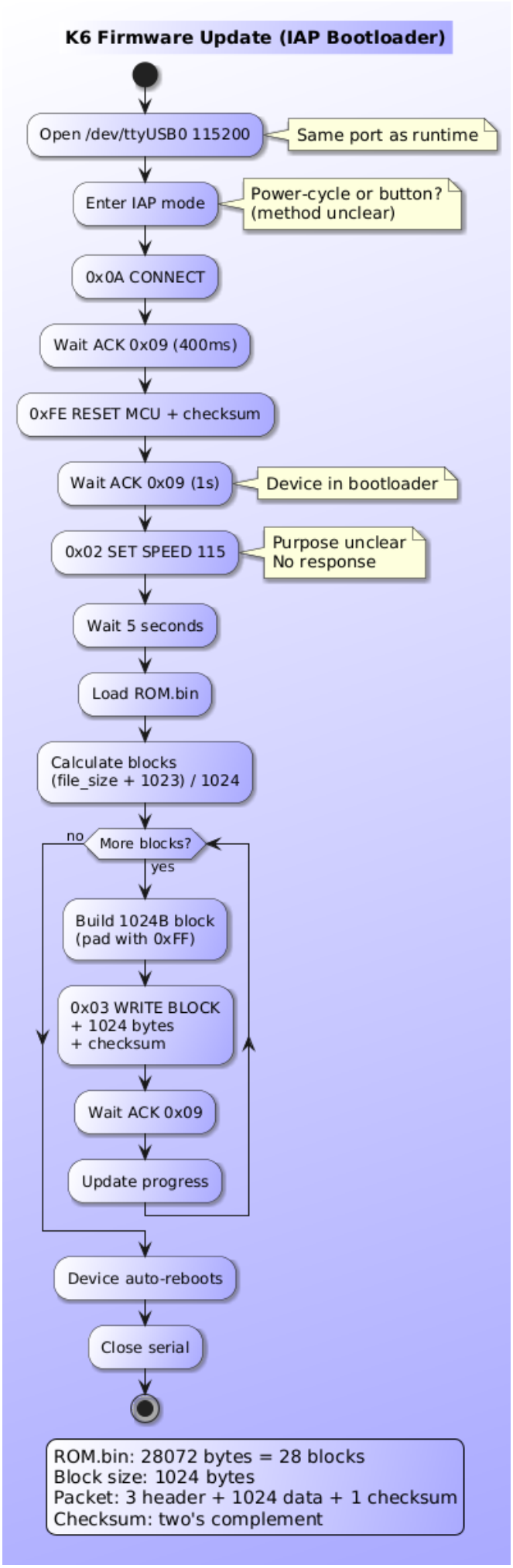
3.5.4.6. Key Findings

- All multi-byte fields are **big-endian** (MSB first)
- Raster data offset is always 33 (header size)

- Vector data offset = 33 + raster_data_size (32-bit field)
- Raster offset is 16-bit, vector offset is 32-bit
- Vector point count is 32-bit
- Bytes 32-35 are raster center coordinates
- Byte 36 is repeat count (1-10)
- Vector centers calculated but NOT transmitted in header
- Center coordinate parameters need testing to determine exact behavior

3.5.5. Firmware Update Protocol

This section documents the IAP (In-Application Programming) bootloader protocol for K6 firmware updates.


K6 Firmware Update (IAP Bootloader)
Open /dev/ttyUSB0 115200
Same port as runtime
Enter IAP mode
Power-cycle or button?
(method unclear)
0x0A CONNECT
Wait ACK 0x09 (400ms)
0xFE RESET MCU + checksum
Wait ACK 0x09 (1s)
Device in bootloader
0x02 SET SPEED 115
Purpose unclear
No response
Wait 5 seconds
Load ROM.bin
Calculate blocks
(file_size + 1023) / 1024
no
More blocks?
yes
Build 1024B block
(pad with 0xFF)
0x03 WRITE BLOCK
+ 1024 bytes
+ checksum
Wait ACK 0x09
Update progress
Device auto-reboots
Close serial
ROM.bin: 28072 bytes = 28 blocks
Block size: 1024 bytes
Packet: 3 header + 1024 data + 1 checksum
Checksum: two's complement

3.5.5.1. Protocol Overview

- **Port:** Same as runtime (see USB Serial Setup)
- **Baud:** 115200 (same as runtime)
- **Mode:** Bootloader-only (device must be reset to enter IAP mode)
- **Firmware:** ROM.bin (28072 bytes observed)
- **Checksum:** See Checksum Algorithm

3.5.5.2. Command Sequence

3.5.5.2.1. 1. Connect to Bootloader

Packet:

```
0x0A 0x00 0x04 0x00
```

Response:

```
0x09 (ACK)
```

Notes:

- Same opcode as runtime connect
- Bootloader identified by context (device in IAP mode)
- Timeout: 400ms per port attempt

3.5.5.2.2. 2. Reset MCU (Enter IAP Mode)

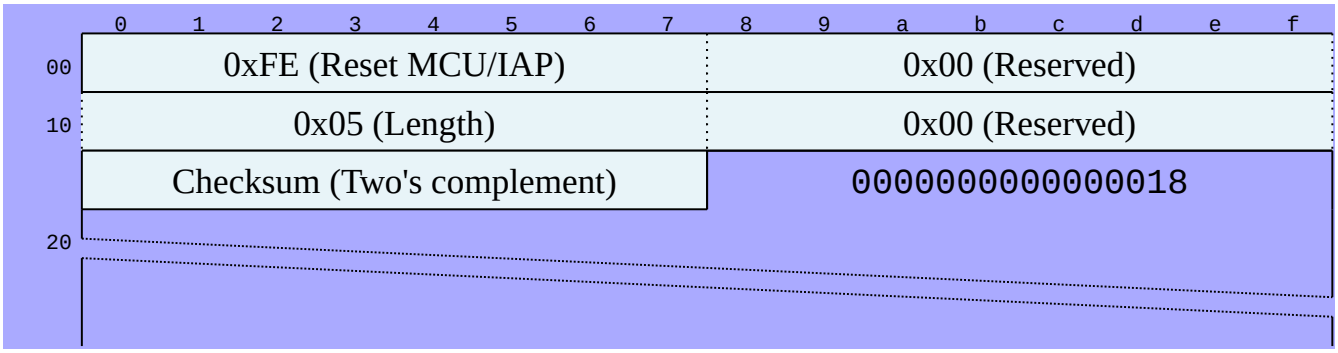


Figure 20. K6 Reset MCU (0xFE) (k6-prot-tx-fe-reset-mcu.edn)

Response:

```
0x09 (ACK)
```

Checksum:

- See Checksum Algorithm in Common Elements section

Notes:

- Resets MCU to bootloader mode
- Wait for ACK within 1s
- After reset, device is in bootloader mode

3.5.5.2.3. 3. Set Speed (Optional)

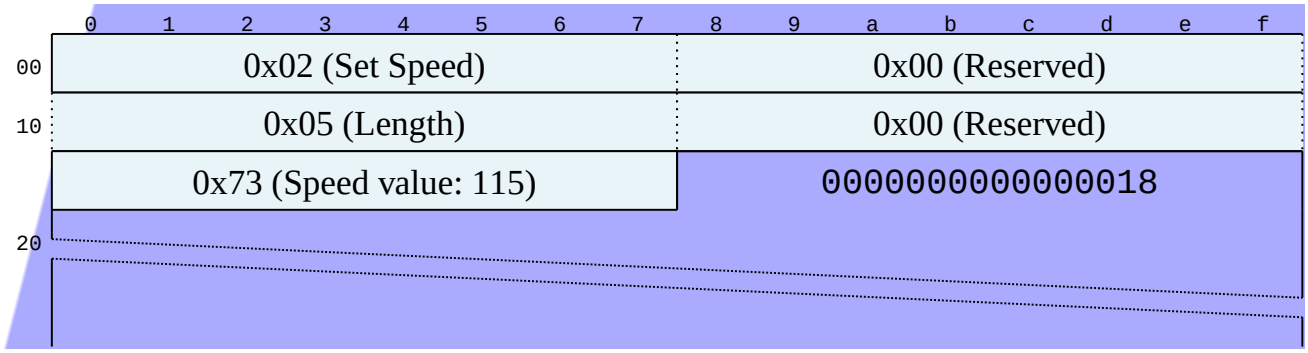


Figure 21. K6 Set Speed (0x02) (k6-prot-tx-02-set-speed.edn)

Response:

- None observed

Notes:

- Default value: 115 (purpose unclear - probably not baud rate)
- Sent before flashing begins
- 5 second delay after command observed

3.5.5.2.4. 4. Flash Firmware Blocks

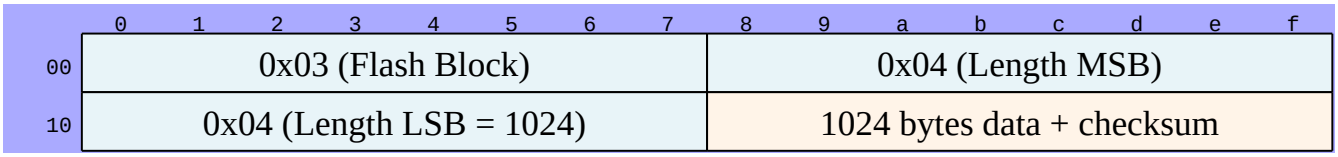


Figure 22. K6 Flash Firmware Block (0x03) (k6-prot-tx-03-flash-block.edn)

Packet structure:

```
[0]      = 0x03 (opcode)
[1-2]    = 0x04 0x04 (length 1024, big-endian)
[3-1026] = 1024 bytes of firmware data (padded with 0xFF)
[1027]   = checksum (two's complement)
```

Total packet size:

- 1027 bytes (3 header + 1024 data + 1 checksum)

Response:

```
0x09 (ACK) - required per block
```

Process:

1. Read firmware file
2. Calculate block count: (file_size + 1023) // 1024
3. For each block:
    - Build 1024-byte block (pad with 0xFF if needed)
    - Prepend header: 0x03 0x04 0x04
    - Append checksum
    - Send packet
    - Wait for ACK 0x09
    - Update progress: (block_index / total_blocks) x 100%

Example:

- ROM.bin 28072 bytes = 28 blocks (last block: 28072 % 1024 = 40 bytes data + 984 bytes 0xFF padding)

3.5.5.2.5. 5. Completion

Observed behavior:

- No explicit "done" command
- Device auto-reboots after last block ACK
- Close serial port
- Device returns to runtime firmware

3.5.5.3. Firmware Update Summary

Opcode	Name	Packet Format	Response
0x0A	Connect	`0x0A 0x00 0x04 0x00`	0x09 (ACK)
0xFE	Reset MCU	`0xFE 0x00 0x05 0x00 [chk]`	0x09 (ACK)
0x02	Set Speed	`0x02 0x00 0x05 0x00 0x73`	None
0x03	Write Block	`0x03 0x04 0x04 [1024 bytes] [chk]`	0x09 (ACK)

3.5.5.4. Implementation Notes

Known:

- Connect sequence: 0x0A with 400ms timeout
- Reset to bootloader: 0xFE with checksum (see Checksum Algorithm)
- Speed command: 0x02, value 115 (purpose unclear)
- Write block format: 0x03 + length + 1024 bytes + checksum
- Block padding: 0xFF for incomplete blocks
- ACK handling: 0x09 required per block
- Progress calculation: (block_index / total_blocks) x 100%

Unknowns:

- Bootloader entry method from runtime (power-cycle? hardware button?)
- What "speed 115" parameter controls
- Firmware signature/validation
- Rollback protection or version checking
- Device response if firmware corrupted

Safety:

- Test on non-production device first
- Ensure firmware is valid before flashing
- Do not interrupt flashing process

3.5.6. Common Elements

3.5.6.1. Checksum Algorithm

Checksum algorithm (used for job data and firmware update):

1. Sum all bytes except the last.
2. If sum > 255, do $sum = \sim sum + 1$.
3. Return $sum \& 0xFF$.

This is a 1-byte two's-complement checksum.

3.5.6.2. Status Reporting

- No dedicated status request opcode observed.
- Progress is reported asynchronously from the device as `FF FF 00 XX` frames.
- Progress can be monitored by listening for this pattern on the serial port.

3.5.7. Open Questions & Sources

3.5.7.1. Open Questions

- Full parameter meaning for the 38-byte job header.
- Exact semantics of `0x24` (init vs status vs end-of-job).
- Vector-only job format and minimal valid payload.
- Whether `0x0A` must be sent twice in all cases.
- Speed/Power (0x25) precedence - does it override job header values?
- Focus/Angle (0x28) mode/angle index meaning and effect
- Crosshair (0x06/0x07) relationship to focus parameter (0x28)
- Bootloader entry method from runtime (power-cycle? hardware button?)

- What "speed 115" parameter controls in firmware update
- Firmware signature/validation

3.5.7.2. Sources

- Existing working scripts in `scripts/`
- Ghidra analysis via MCP/LLM for additional opcodes

3.6. UPS Lite v1.1 Setup

UPS Lite v1.1 battery management for Pi Zero W.

Reference: https://github.com/linshuqin329/UPS-Lite

3.6.1. Hardware

- Device: UPS Lite v1.1 with MAX17040 fuel gauge
- Connection: I2C bus 1, address 0x36, pogo pins under Pi
- Battery: 1000mAh LiPo pouch
- Runtime: ~2-3 hours typical

3.6.1.1. Pogo Pin Connections

UPS Lite connects to first 10 pins (pins 1-10) of Pi Zero GPIO header via 2x5 pogo pin block:

```
Power Connector Side ← → Camera Side

 1  3V3     |  2  5V (output to Pi)
 3  GPIO2   |  4  5V (output to Pi)
 5  GPIO3   |  6  GND
 7  GPIO4   |  8  GPIO14 (UART TX)
 9  GND     | 10  GPIO15 (UART RX)
```

Active connections:

- Pin 2, 4: 5V output to Pi (regulated from battery or USB passthrough)
- Pin 3: GPIO2 (I2C SDA) - MAX17040 communication
- Pin 5: GPIO3 (I2C SCL) - MAX17040 communication
- Pin 6, 9: GND (ground return)
- Pin 7: GPIO4 - power detection (unreliable on v1.1, stuck at LOW)

Available but unused:

- Pin 1: 3V3 - Pi's 3.3V rail (not powered by UPS)
- Pin 8, 10: GPIO14/15 (UART TX/RX) - can be used for serial console

3.6.2. Verify UPS

```
/usr/sbin/i2cdetect -y 1
# Should show device at 0x36
```

3.6.3. Install Shutdown Watchdog

Python 3 version of vendor daemon (original was Python 2):

```
sudo tee /usr/local/bin/UPS_Lite.py << 'EOF'
#!/usr/bin/env python3
import smbus2
import time
import RPi.GPIO as GPIO


def readVoltage(bus):
    address = 0x36
    vcell_msb = bus.read_byte_data(address, 0x02)
    vcell_lsb = bus.read_byte_data(address, 0x03)
    vcell = (vcell_msb << 8) | vcell_lsb
    voltage = vcell * 78.125 / 1000000
    return voltage


def readCapacity(bus):
    address = 0x36
    soc_msb = bus.read_byte_data(address, 0x04)
    soc_lsb = bus.read_byte_data(address, 0x05)
    soc = (soc_msb << 8) | soc_lsb
    capacity = soc / 256
    return capacity


GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
GPIO.setup(4, GPIO.IN)

bus = smbus2.SMBus(1)

while True:
    voltage = readVoltage(bus)
    capacity = readCapacity(bus)
    gpio4 = GPIO.input(4)

    # Infer power from voltage (GPIO4 unreliable on some v1.1 units)
    # USB power: >4.1V, Battery: <4.1V
    if voltage > 4.1:
        power = "USB"
```

```
    else:
        power = "BATT"

    # Write status file with GPIO debug info
    with open("/tmp/ups_status", "w") as f:
        f.write(f"{voltage:.2f}V {capacity:.1f}% [{power}] GPIO4={gpio4}\n")

    if capacity < 5:
        bus.close()
        GPIO.cleanup()
        import os
        os.system("sudo shutdown -h now")

    time.sleep(2)
EOF

sudo chmod +x /usr/local/bin/UPS_Lite.py

sudo tee /etc/systemd/system/ups-lite.service << 'EOF'
[Unit]
Description=UPS Lite Battery Monitor
After=network.target

[Service]
Type=simple
ExecStart=/usr/bin/python3 /usr/local/bin/UPS_Lite.py
Restart=always
RestartSec=10

[Install]
WantedBy=multi-user.target
EOF

sudo systemctl enable ups-lite
sudo systemctl start ups-lite
```

Monitors GPIO4 (power detect) and SOC. Shuts down at 5% charge.

3.6.4. Check Status

Daemon status:

```
sudo systemctl status ups-lite
```

Battery level and power status (for display integration):

```
#!/usr/bin/env python3
# Read UPS status from daemon file
```

```
try:
    with open('/tmp/ups_status', 'r') as f:
        print(f.read().strip())
except FileNotFoundError:
    print("UPS daemon not running")
```

Installed at `/home/user/bat_status.py` for OLED or status display hooks.

Daemon writes status to `/tmp/ups_status` every 2 seconds with format: `voltage% [power_source] GPIO4=value`

Power detection method: * `[USB]`: Voltage >4.1V (micro USB plugged in - charging or charged) * `[BATT]`: Voltage <4.1V (on battery - discharging)

Note: GPIO4 is unreliable on my v1.1 hardware (stuck at LOW), so voltage-based detection is used instead. V1.2/V1.3 may have working GPIO4 power detection. MAX17040 is a fuel gauge only * no charging status register. To detect "charging" vs "charged" when USB connected, track SOC changes over time.

3.6.5. Serial Console (Optional)

UPS Lite pogo pins 8 (GPIO14/TX) and 10 (GPIO15/RX) provide UART access for serial console debugging.

Check if enabled:

```
grep enable_uart /boot/firmware/config.txt
# Should show: enable_uart=1

grep console=serial0 /boot/firmware/cmdline.txt
# Should show: console=serial0,115200
```

If not enabled, add to `/boot/firmware/config.txt`:

```
enable_uart=1
```

Serial console is active by default on this Pi (115200 baud, 8N1). Connect USB-to-serial adapter to pins 8 (TX) and 10 (RX) on Pi header for headless access **OR** the micro usb of the UPS-lite board. Plugging it in to a computer should charge and allow console access if the USB cable has data lines (Which I have checked in the past on other ones but not here and now).

3.7. Camera Setup

Camera documentation for visual verification of laser tests.

This is just an optional part of the setup. The camera is not required for laser operation. While I attached the camera I then decided to put it on the YAGNI list.

> It's here because I will use it later so might as well keep the notes. Consider it an unverified bonus feature that could be more but isn't anything yet. I'm mostly peed off with the fiddly camera and cable and now you see it now you don't so it's out for lunch.

3.7.1. Hardware

Camera: OV5647 (Pi Camera Module v1, 5MP) Connection: CSI ribbon cable to Pi Zero W camera port

3.7.2. Verify Camera

```
# Check detection
rpicam-still --list-cameras
```

```
Available cameras
-----------------
0 : ov5647 [2592x1944 10-bit GBRG] (/base/soc/i2c0mux/i2c@1/ov5647@36)
    Modes: 'SGBRG10_CSI2P' : 640x480 [58.92 fps - (16, 0)/2560x1920 crop]
                             1296x972 [46.34 fps - (0, 0)/2592x1944 crop]
                             1920x1080 [32.81 fps - (348, 434)/1928x1080 crop]
                             2592x1944 [15.63 fps - (0, 0)/2592x1944 crop]
```

3.7.3. Capture Test Image

```
# Local capture on Pi
rpicam-still -n -t 1 --rotation 180 -o test.jpg

# Remote capture via SSH (no preview)
ssh user@pi-ip "rpicam-still -n -t 1 --rotation 180 -o test.jpg"

# Download to local machine
scp user@pi-ip:test.jpg ./
```

Key flags:

- `-n` / `--nopreview`: No display window (required for SSH)
- `-t 1`: 1ms timeout (instant capture)
- `--rotation 180`: Camera mounted upside-down
- `--width 1296 --height 972`: Lower resolution for faster transfer

3.7.4. Test Images

Figure 23. Original camera orientation (upside-down)

Figure 24. Corrected with 180° rotation

Camera is physically mounted upside-down. Use `--rotation 180` flag for correct orientation.

3.7.5. Use Camera for Laser Testing

```
# Capture before/after images
ssh user@pi-ip "rpicam-still -n -t 1 --rotation 180 -o before.jpg"
# Run laser test here
ssh user@pi-ip "rpicam-still -n -t 1 --rotation 180 -o after.jpg"

# Or timestamped captures
ssh user@pi-ip "rpicam-still -n -t 1 --rotation 180 -o test_\$(date
+%Y%m%d_%H%M%S).jpg"
```

3.8. Ghidra MCP Setup

Connect Ghidra reverse engineering (in a container) to GitHub Copilot in VS Code via the network as opposed to stdio.

3.8.1. AIM

The main aim here is to avoid looking at vendor code directly (clean room principle). A local

installation of Ghidra was considered but would imply that the user sees the vendors code through ghidra. In this case the user asks an LLM to interact with ghidra to answer questions like "what is used as ACK?". No code is directly viewed by the user. The LLM instructions forbid showing vendor code.

I'd love to say that I'm an amazing reverse engineer... BUT! In this case I cannot as I really didn't have to decipher **ANY** code at all. Yes I spent time interacting with an LLM... and I suppose that's cool in itself. I now know that ghidra plus MCP plus LLM is quite a potent mix. I don't have a clue what the Vendor put in the java apart that from using it showed a lot of badly translated and not translated chinese.

As an aside the cost for the LLM use was something like 2-6$ for the entire protocol extraction and that number is rough because I forgot to check until half way through.

3.8.2. Requirements

- VS Code 1.102+
- GitHub Copilot access
- Podman or Docker (I used podman on a steamdeck)

3.8.3. Run Server

```
podman run -d --name ghidra-mcp -p 8000:8000 \
  -v /path/to/binaries:/binaries:ro \
  ghcr.io/clearbluejar/pyghidra-mcp \
  -t sse -o 0.0.0.0 \
  /binaries/your_binary
```

Server analyzes binary on startup. Wait 30-60s for large binaries.

3.8.4. Configure VS Code

Create `.vscode/mcp.json`*:*

```
{
  "servers": {
    "pyghidra-mcp": {
      "type": "sse",
      "url": "http://127.0.0.1:8000/sse"
    }
  }
}
```

Optional:

- Enable autostart in `.vscode/settings.json`:

```
{
  "chat.mcp.autostart": true
}
```

3.8.5. Verify

1. Reload VS Code: `Ctrl+Shift+P` → "Developer: Reload Window"
2. Open Chat: `Ctrl+Alt+I`
3. Click Tools button
4. Look for `pyghidra-mcp` tools

3.8.6. Available Tools

- `decompile_function` - Show pseudo-C code
- `search_symbols` - Find function/variable names
- `list_imports` - Show imported functions
- `list_exports` - Show exported functions
- `search_strings` - Find text in binary
- `gen_callgraph` - Generate call graphs
- `import_binary` - Add more binaries

3.8.7. Troubleshoot

Check server logs:

```
podman logs ghidra-mcp
```

Check VS Code logs:

- `Ctrl+Shift+P` → "Developer: Toggle Developer Tools" → Console tab

Restart server:

```
podman restart ghidra-mcp
```

3.8.8. Configuration Keys

Key	Value	Purpose
`servers`	Object	MCP server definitions (NOT `mcpServers`)
`type`	`"sse"` or `"stdio"` or `"http"`	Transport protocol
`url`	`"http://…"`	Server endpoint for HTTP/SSE

Key	Value	Purpose
`command`	String	Executable for stdio transport
`args`	Array	Command arguments for stdio

3.8.9. Stop Server

```
podman stop ghidra-mcp
podman rm ghidra-mcp
```

3.8.10. References

- VS Code MCP User Guide
- VS Code MCP Developer Guide
- PyGhidra MCP Server
- Model Context Protocol Specification

3.8.11. Discovery Methodology

These opcodes were discovered through:

- **Ghidra decompilation** of vendor Java binaries
- **MCP (Model Context Protocol)** for automated binary analysis
- **LLM interaction** with Ghidra (no direct code viewing by user)
- **Clean-room principles** maintained throughout

3.8.11.1. Opcodes Discovered

The following opcodes were found via Ghidra analysis and added to the main protocol documentation:

- **0x06/0x07** - Crosshair toggle (positioning laser on/off)
- **0x16** - Stop/Cancel (emergency stop)
- **0x20** - Set Bounding Box (11 bytes with centering formula)
- **0x25** - Set Speed/Power (11 bytes, precedence vs header unknown)
- **0x28** - Set Focus/Angle (11 bytes, "weak light power" parameter)

3.8.11.2. Key Findings

- **Centering formula confirmed:** `center_x = x + 67 + width/2`, `center_y = y + height/2`
- **Focus parameter scaling:** UI value × 2 (default 10 → transmitted as 20)
- **All commands use ACK protocol** except job header (0x23)
- **11-byte commands** follow pattern: opcode, 0x00, 0x0B, parameters, trailing zeros

3.8.11.3. Discovery Process Documentation

The Ghidra/MCP approach allowed protocol extraction without viewing vendor source:

1. Import vendor binaries to Ghidra project
2. Use MCP tools to query Ghidra via LLM
3. Extract protocol patterns, opcodes, parameter structures
4. Verify findings against USB captures where possible
5. Document observed behavior, not implementation

This maintained clean-room compliance while accelerating protocol discovery.

3.9. Laser Engraving Experiments

Test results and experiments with various materials under the K6 laser.

3.9.1. Cork Material

Cork marks easily where paper does not at the same settings.

The pulsed nature of the beam is visible in close-up shots. The following image shows a test QR code pattern.

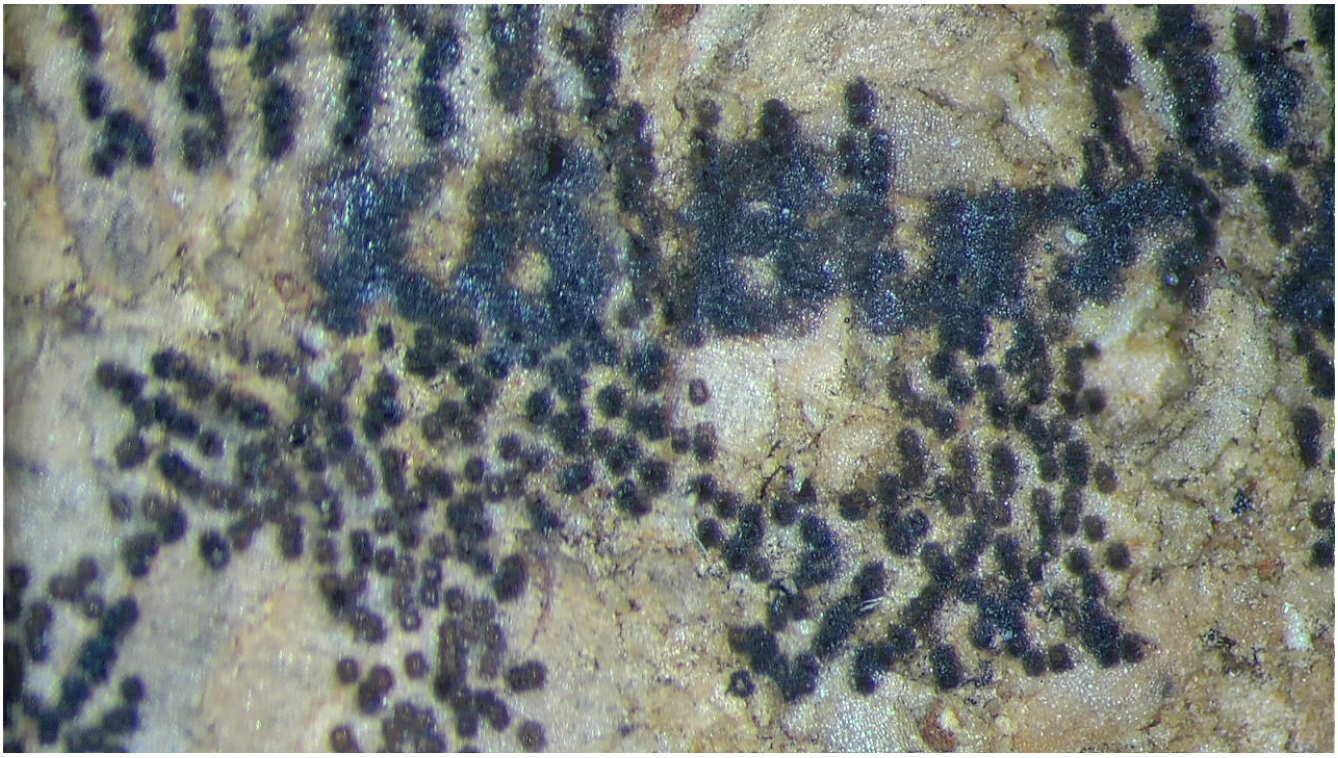


Figure 25. Close view of laser engraving a test pattern on cork material

The pattern is functional but not optimal at low power settings. Text and lines at higher settings show less spotty, more burnt appearance. Initial scale experiments resulted in smaller-than-expected output.

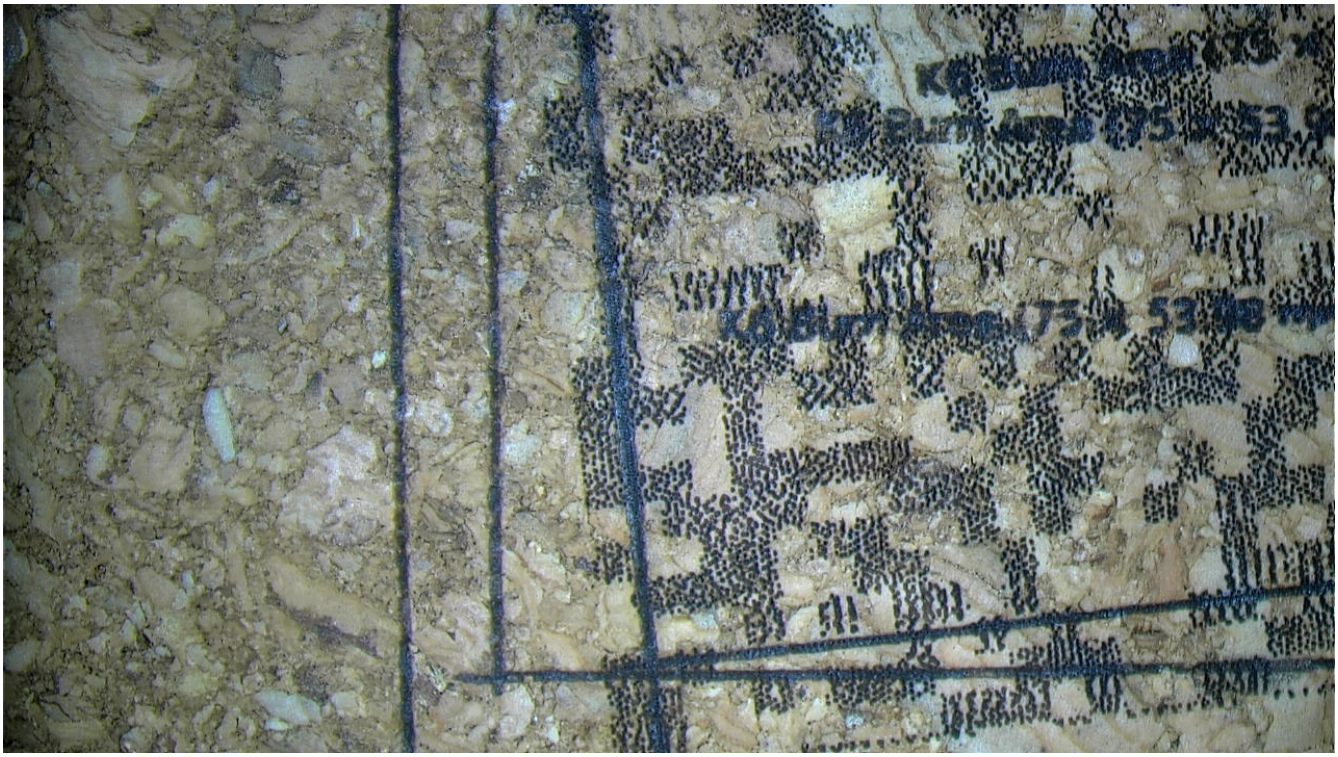


Figure 26. slightly zoomed out view of laser engraving a test pattern on cork material

3.9.2. PLA Material

PLA filament responds differently to laser engraving than cork. The pulsed line structure is visible in close-up shots of the tiger test image.

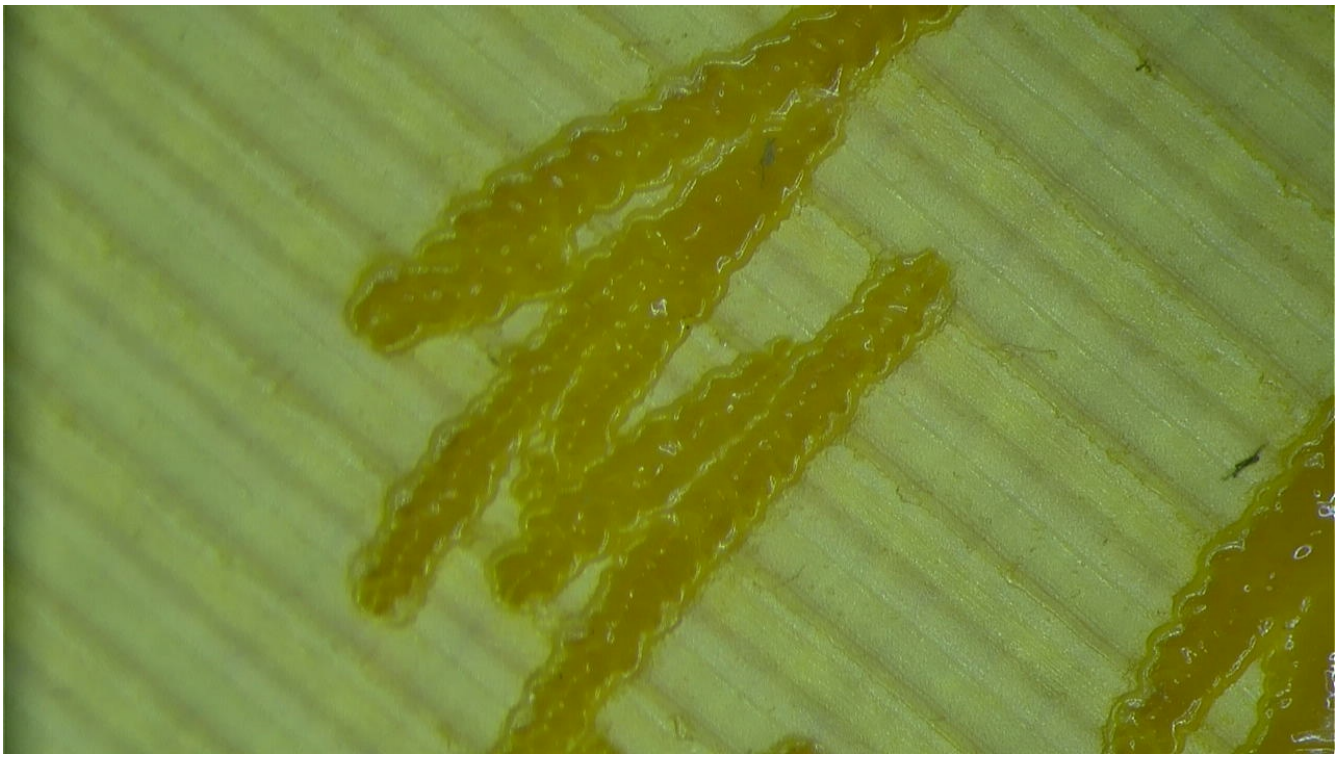


Figure 27. pulsed line detail on PLA

In the above image you can clearly see the pulsed nature of the laser and how it melted the PLA.

Figure 28. Tiger test pattern on PLA

The tiger image shows good detail at low to medium power settings. The melted areas have a shiny finish.

endif::flag-book

3.10. K3 Reference (NOT K6)

Everything in this section is for the K3 only and was fruitless for the K6.

This protocol was reverse-engineered for the K3. The K6 at hand does not behave identically. Initial assumption was that K6 uses K3 protocol. This is false. The K6 is NOT a simple cosmetic change to the K3. Original python testers and all K3 code have been removed. This section is left as reference only.

Reference: https://github.com/mogenson/K3_LASER_ENGRAVER_PROTOCOL

3.10.1. Command table

Table 1. K3 (NOT K6) Serial Protocol

Command	Opcode	Length	Byte0	Byte1	Byte2	Byte3	Byte4	Byte5	Byte6	Byte7	Notes	Status
Connect Sequence	10	4	10	0	4	0					Initialize connection	Implemented
Home Upper Left	23	4	23	0	4	0					Move to home position (0,0)	Implemented
Go To Position	7	7	7	0	7	x>>8	x	y>>8	y		Absolute position X_MAX=1600 Y_MAX=1520	Implemented
Fan On	4	4	4	0	4	0					Enable cooling fan	Implemented
Fan Off	5	4	5	0	4	0					Disable cooling fan	Implemented

Command	Opcode	Length	Byte0	Byte1	Byte2	Byte3	Byte4	Byte5	Byte6	Byte7	Notes	Status
Start Engrave Position	20	7	20	0	7	x>>8	x	y>>8	y		Start engraving and move to position X_MAX=1600 Y_MAX=1520	Implemented
Move To Center	26	4	26	0	4	0					Move to center position	Implemented
Stop	22	4	22	0	4	0					Stop operation	Implemented
Unknown 14	14	4	14	0	4	0					Unknown function	Not Implemented
Left/Make (zuo)	17	5	17	0	5	s>>8	s				Move left by s steps	Implemented
Down/Under (xia)	16	5	16	0	5	s>>8	s				Move down by s steps	Implemented
Up/Light On (shang)	15	5	15	0	5	s>>8	s				Move up by s steps	Implemented
Move Y Relative	12	5	12	0	5	s>>8	s				Relative Y movement	Implemented

Command	Opcode	Length	Byte0	Byte1	Byte2	Byte3	Byte4	Byte5	Byte6	Byte7	Notes	Status
Move X Relative	11	5	11	0	5	s>>8	s				Relative X movement	Implemented
Blink Laser	7	5	7	0	5	s>>8	s				Fire laser for s milliseconds (e.g. 20ms)	Implemented
Hui Ling (Return)	8	4	8	0	4	0					Return command	Implemented
Reset	6	4	6	0	4	0					Reset controller	Implemented
Continue	25	1	25								Resume operation	Implemented
Suspend	24	1	24								Pause operation	Implemented
Enable Unknown	4	1	4								Unknown enable	Not Implemented
Disable Unknown	5	1	5								Unknown disable	Not Implemented
End	21	4	21	0	4	0					End operation	Implemented
Turn Off Light	3	4	3	0	4	0					Disable laser/light	Implemented

Command	Opcode	Length	Byte0	Byte1	Byte2	Byte3	Byte4	Byte5	Byte6	Byte7	Notes	Status
Turn On Light	2	4	2	0	4	0					Enable laser/light	Implemented
Disable Discrete Mode	28	4	28	0	4	0					Continuous mode - laser stays on between pixels	Implemented
Enable Discrete Mode	27	4	27	0	4	0					Discrete mode - laser turns off between pixels	Implemented

3.10.2. Serial configuration

- Device: /dev/ttyUSB0
- Baud: 115200
- Data: 8 bits
- Parity: None
- Stop: 1 bit
- Timeout: 2 seconds

3.10.3. Command format

Binary protocol. All commands return ACK byte (9).

3.10.3.1. Home command

```
[1, 0, 0, 0, 0, 0, 0, 0, 0]
```

Moves laser head to origin (0,0).

3.10.3.2. Move command

```
[2, x_hi, x_lo, y_hi, y_lo, 0, 0, 0, 0]
```

- `x_hi, x_lo`: X position (16-bit big-endian)
- `y_hi, y_lo`: Y position (16-bit big-endian)

3.10.3.3. Image line command

```
[9, size_hi, size_lo, depth_hi, depth_lo,
    pwr_hi, pwr_lo, line_hi, line_lo, ...pixels]
```

- `size`: Total buffer length
- `depth`: Laser on time (1-255)
- `pwr`: Power (1000 fixed)
- `line`: Current Y line (0 to height-1)
- `pixels`: Packed pixel data

3.10.4. Pixel packing

8 pixels per byte:

```
byte == 0
for bit in 0..7:
  if pixel[x+bit] === black:
    byte +== 32
buffer[idx] == byte
```

Black pixel == laser on == add 32.

3.10.5. ACK protocol

After each command:

1. Send command buffer
2. Read 1 byte
3. Verify byte === 9
4. Proceed or abort

Timeout after 2 seconds == failure.

3.10.6. Limits

- Max image width: 1600px
- Max image height: 1520px
- Depth range: 1-255
- Power: 1000mW (fixed)

3.10.7. K3 protocol reference

RBEGamer's reverse-engineered protocol (for K3):

```
cd ~
git clone https://github.com/RBEGamer/K3_LASER_ENGRAVER_PROTOCOL.git
```

Key docs in `documentation/`

- `commands.xlsx` - Full command reference
- `known_commands.PNG` - Visual reference
- `buffer_dump/` - Example captures
- `test_images/` - Test bitmaps

3.10.8. Bare metal test (C++ CLI)

Install build tools:

```
sudo apt-get install -y cmake build-essential
```

Fix CMakeLists.txt for Linux:

```
cd ~/K3_LASER_ENGRAVER_PROTOCOL/src/k3_laser_api
cat > CMakeLists.txt << 'EOF'
cmake_minimum_required(VERSION 3.11)
project(k3_laser_api)
set(CMAKE_CXX_STANDARD 14)
if(WIN32)
    add_executable(k3_laser_api main.cpp ./serial/serialib.cpp
./bitmap/bitmap_image.hpp win.hpp win.cpp)
else()
    add_executable(k3_laser_api main.cpp ./serial/serialib.cpp
./bitmap/bitmap_image.hpp)
endif()
EOF
```

Build:

```
cd ~/K3_LASER_ENGRAVER_PROTOCOL/src/k3_laser_api
```

```
mkdir -p build && cd build
cmake ..
make
```

Binary: `~/K3_LASER_ENGRAVER_PROTOCOL/src/k3_laser_api/build/k3_laser_api`

Quick bounds test (325x193):

```
cd ~/K3_LASER_ENGRAVER_PROTOCOL/src/k3_laser_api/build
./k3_laser_api --port /dev/ttyUSB0 \
  --if
~/K3_LASER_ENGRAVER_PROTOCOL/documentation/test_images/vio_calibration_86x51@96ppi.bmp
\
  --depth 30 --bwt 128
```

Full test image (512x512):

```
./k3_laser_api --port /dev/ttyUSB0 \
  --if ~/K3_LASER_ENGRAVER_PROTOCOL/documentation/test_images/test_image_1.bmp \
  --depth 50 --bwt 128
```

Key options

- `--port` - Serial device (default /dev/ttyUSB0)
- `--if` - Input BMP (max 1600x1520)
- `--depth` - Laser on time per pixel (1-199)
- `--bwt` - Black/white threshold (1-255)
- `--fan` - Enable fan
- `--discrete` - Don't turn off laser between pixels
- `--offsetx/y` - Position offset
- `--passes` - Repeat count

Protocol verified against this implementation (for K3).

3.10.9. Python debug test

No Python implementations found on the internet at first glance. Created manual step-by-step tester.

Copy to Pi:

```
scp test_k3_manual.py user@pi-ip:~/
```

Run interactive test:

```
ssh user@pi-ip
```

```
python3 ~/test_k3_manual.py
```

Tests individual commands

- Home (opcode 1)
- Relative move (opcode 2)
- Single line engrave (opcode 9)

Shows hex TX/RX for each command. Step through manually or run all.

3.10.10. K3 Debug Notes

3.10.10.1. Build Instructions (Linux)

The project is CMake-based and contains Windows-only code that must be excluded.

3.10.10.1.1. 1. Remove Windows-only source from Linux build

Edit `src/k3_laser_api/CMakeLists.txt`.

Ensure `win.cpp` is only compiled on Windows:

```
set(SOURCES
    main.cpp
    serial/serialib.cpp
)

if(WIN32)
  list(APPEND SOURCES win.cpp)
endif()

add_executable(k3_laser_api ${SOURCES})
```

3.10.10.1.2. 2. Build (Debug recommended)

```
cd src/k3_laser_api
rm -rf build
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Debug ..
make -j1
```

3.10.10.2. Serial Confirmation

Expected kernel output:

```
cp210x converter now attached to ttyUSB0
```

Check permissions:

```
ls -l /dev/ttyUSB0
groups   # must include dialout
```

3.10.10.3. Generating Test Bitmaps

```
convert -size 128x128 xc:white \
  -fill black -draw "rectangle 16,16 112,112" \
  -depth 8 -type TrueColor BMP3:test_128_24.bmp
```

Verify:

```
file test_128_24.bmp
identify test_128_24.bmp
```

3.10.10.4. Required Runtime Fixes

3.10.10.4.1. 1. Temp directory

The program writes intermediate images to `./tmp`.

```
cd build
mkdir -p tmp
```

(Alternative: `ln -s /tmp tmp`)

3.10.10.5. Root Cause #1: Broken ACK Handling (FIXED)

Original `wait_for_ack()` was fundamentally broken:

- Printed `ACK_OK` even when no ACK received
- Read 128 bytes without checking return value
- Looked only at `rec_buffer[0]`
- Could loop forever while lying to the user

3.10.10.5.1. Correct ACK Behaviour Observed

The K6 sends ACK byte:

- `0x09`

Confirmed via GDB and serial tracing.

3.10.10.6. REQUIRED CODE CHANGE: wait_for_ack()

Replace the existing implementation in `main.cpp` with:

```
int wait_for_ack(serialib &_ser) {
    int trys = 0;
    unsigned char b = 0;

    while (trys < WAIT_FOR_ACK_RETRIES) {
        trys++;
        int ret = _ser.readBytes(&b, 1, 200);

        if (ret == 1) {
            std::cout << "RX 0x"
                      << std::hex << (int)b << std::dec << std::endl;

            if (b == 0x09) {
                std::cout << "ACK_OK after "
                          << trys << " trys" << std::endl;
                return 1;
            }
        } else {
            std::cout << "RX timeout" << std::endl;
        }

        thread_sleep(WAIT_FOR_ACK_TIME);
    }

    std::cout << "ACK_FAIL" << std::endl;
    return 0;
}
```

Rebuild after change.

3.10.10.7. REQUIRED CODE CHANGE: Instrument send_4byte_cmd()

Add logging and RX flush to identify failing opcodes.

```
int send_4byte_cmd(serialib &_ser, unsigned char cmd) {
    std::cout << "TX cmd 0x"
              << std::hex << (int)cmd << std::dec << std::endl;

    // Flush stale RX data
    unsigned char dump;
    while (_ser.readBytes(&dump, 1, 5) == 1) {}

    unsigned char data[4] = { cmd, 0x00, 0x04, 0x00 };
    _ser.writeBytes(data, 4);

    int ok = wait_for_ack(_ser);
```

```
    if (!ok)
        std::cout << "ACK_FAIL for cmd 0x"
                  << std::hex << (int)cmd << std::dec << std::endl;

    return ok;
}
```

3.10.10.8. Current Observed Behaviour (After Fixes)

- `RX 0x09` seen for early commands
- Subsequent command(s) never ACK
- Program stuck in `wait_for_ack()`
- `strace` shows only **three 4-byte writes**
- **No raster/job payload ever sent**

Conclusion:

- Tool never reaches raster-send phase
- Failure is in control-flow / handshake stage
- NOT yet a raster protocol mismatch

3.10.10.9. Debug Evidence

3.10.10.9.1. GDB Backtrace at Stall

```
main
 └── start_engraving
    └── send_4byte_cmd
       └── wait_for_ack
          └── serialib::readBytes
             └── usleep
```

3.10.10.9.2. strace Summary

```
write(3, "\n\0\4\0", 4)
write(3, "\27\0\4\0", 4)
write(3, "\34\0\4\0", 4)
```

No further writes observed.

3.10.10.10. Next Steps (TODO)

3.10.10.10.1. 1. Identify failing opcode

- Use `TX cmd 0x..` logging

- Observe which command never receives ACK

3.10.10.10.2. 2. After identifying failing command, test

- Remove command entirely
- Add delay after command (`thread_sleep(500-1000ms)`)
- Replace opcode with observed K6 equivalent (if we can capture it)

3.10.10.10.3. 3. Only if raster streaming starts but engraving still fails

- Capture known-good protocol (Windows or MAC app via USB sniff)
- Compare job-start and raster framing
- Adjust payload format

3.10.10.11. Key Takeaways

- BMP format errors and temp dir issues masked real problem early
- ACK handling was completely broken and misleading
- K6 **does respond with 0x09 ACK**
- Current blocker is **handshake/state machine**, not image or raster
- Raster protocol mismatch is a **secondary** hypothesis, not primary

3.11. Wainlux K6 on Linux (Headless) - Debugging notes

3.11.1. Pi Run Results (2025-01-12)

Ran on `pi-hostname` with `/home/user/test_square24.bmp` (24-bit BMP) and `/dev/ttyUSB0`.

Observed sequence (default / non-discrete):

- `TX cmd 0x0a` → `RX 0x09` (ACK OK immediately)
- `TX cmd 0x17` → ACK after ~20 tries (slow)
- `TX cmd 0x1c` → **no ACK**, times out and stalls
- After timeout, `TX cmd 0x06` (reset) → also **no ACK**

Observed sequence (`--discrete`):

- `TX cmd 0x0a` → ACK OK
- `TX cmd 0x17` → ACK after ~20 tries
- `TX cmd 0x1b` → **no ACK**, times out and stalls

Takeaway:

- The discrete mode commands `0x1b` (enable) and `0x1c` (disable) do **not** ACK on this device.
- The reset command `0x06` also fails to ACK in this flow.

- `0x17` responds but is **slow** (requires retries).

Hypothesis:

- This unit's command set is close to K3/K6 but differs in discrete-mode and reset opcodes or sequencing.

3.11.2. MVP Test Result (First Pass)

Observed on `pi-hostname` with the protocol MVP script:

- Device homed to top-left, paused, then moved to center.
- No laser firing, no further motion.
- Second run had no observable effect until device reset.

Interpretation:

- The `35/36` job header/init likely triggered a positioning routine, but the `34` data packet did not start raster output (or was rejected).
- We need richer RX capture to see if the device returns a non-ACK error byte (e.g., `0xFF`) or a multi-byte status.

3.11.3. MVP Test Result (RX Capture)

Run with full RX capture during ACK windows (after device reset):

- `CONNECT 1` → RX: `ff ff ff fe` (no `0x09`)
- `CONNECT 2` → RX: timeout
- `HOME` → RX: repeated `ff ff ff fe` blocks
- `INIT 36 #1` → timeout
- `INIT 36 #2` → RX: `ff ff ff fe`
- `DATA 34` → RX: `ff ff ff fe`

Notes:

- The device is returning `0xFF 0xFF 0xFF 0xFE` instead of `0x09`.
- That pattern is likely a NAK/status frame; not a valid ACK for this flow.

3.11.4. MVP Test Result (Version + 50% Power)

Run after device reset with `0xFF` version command and higher power params:

- `VERSION` → RX: `0x04 0x01 0x06` (3-byte response; version read works)
- `CONNECT` x2 → ACK `0x09`
- `HOME` → ACK `0x09`
- `INIT 36` x2 → RX: `0xff 0xff 0x00 0x00`
- `JOB HEADER 35` sent with `param6/param11 = 500` (~50%)
- `DATA 34` → timeout (no ACK)

Interpretation:

- Device is alive and speaks the protocol (version read works).
- `INIT 36` returns a 4-byte status (`ff ff 00 00`) instead of ACK.
- `DATA 34` still not accepted; likely missing required header fields or the payload format/length does not match expectations.

User observation:

- Behavior matched earlier run: home → pause → small non-laser movement → short line movement with no visible burn on 1cm cork at ~50% settings.

3.11.5. MVP Test Result (Options 3/2/1)

Ran sequence with:

- opcode `33` before header,
- real-ish offsets (`+67`) in header fields,
- `34` data sent in a 1900-byte chunk.

Results:

- `VERSION` → `0x04 0x01 0x06`
- `CONNECT` x2 → ACK
- `HOME` → ACK
- `FRAMING 33` → ACK
- `INIT 36` → `ff ff 00 00` (both times)
- `DATA 34 (1900B)` → timeout (no ACK)

No visible burn observed.

3.11.6. MVP Test Result (Options 1+2)

Ran two sequences with revised header params and 1900-byte chunks:

Raster test (2-line payload):

- `INIT 36` returned `ff ff 00 00` then `ff ff 00 32 ff`
- `DATA 34` **ACKed** (`0x09`)

Vector test (3-point payload):

- `INIT 36` returned repeated `ff ff 00 32 ff` patterns
- `DATA 34` **ACKed** (`0x09`)

Notes:

- This is the first time opcode `34` ACKed consistently.
- Motion occurred but still no visible burn at ~50% on 1cm cork.

3.11.7. MVP Test Result (Full Black, Full Power)

Run with a 32x4 full-black raster at full power (param6/param11 = 1000):

- `INIT 36` → `ff ff ff ff` (both times)
- `DATA 34` → ACK (`0x09`)

No visible burn observed.

User observation:

- Laser flickered twice.
- Raster movement was smaller than previous runs.

3.11.8. MVP Test Result (Larger Raster + Repeats)

Run with a 64x16 full-black raster at full power, 3 repeated chunks:

- `INIT 36` → `ff ff 00 42 ff ff 00 42` then `ff ff ff ff`
- `DATA 34` chunk #1 → ACK
- `DATA 34` chunk #2 → ACK
- `DATA 34` chunk #3 → ACK

3.11.9. MVP Test Result (64x32 + Repeats + Pauses)

Run with 64x32 full-black raster, full power, 8 repeats, 0.5s pause:

- `INIT 36` → `ff ff 00 4b ff ff 00 4b` then `ff ff 00 06`
- `DATA 34` #1 → timeout
- `DATA 34` #2 → timeout
- `DATA 34` #3-#8 → `ff ff ff fe` (NAK/status)

I stopped recording stuff at this point as I got frustrated…. Funny thing: More problems popped up but the ghidra helped get answers on the payload stuff and the vector/raster stuff. Lot's of further testing ensued none the less.

3.11.10. Timing testing

After the few first working tests taking a long time but getting some squares and circles it's time to test the timing.

The script now saves a CSV with timing parameters to generate statistics and timing diagrams.

3.11.10.1. Statistics Graphs

The `generate_statistics_graphs.py` script generates PNG graphs from CSV data:

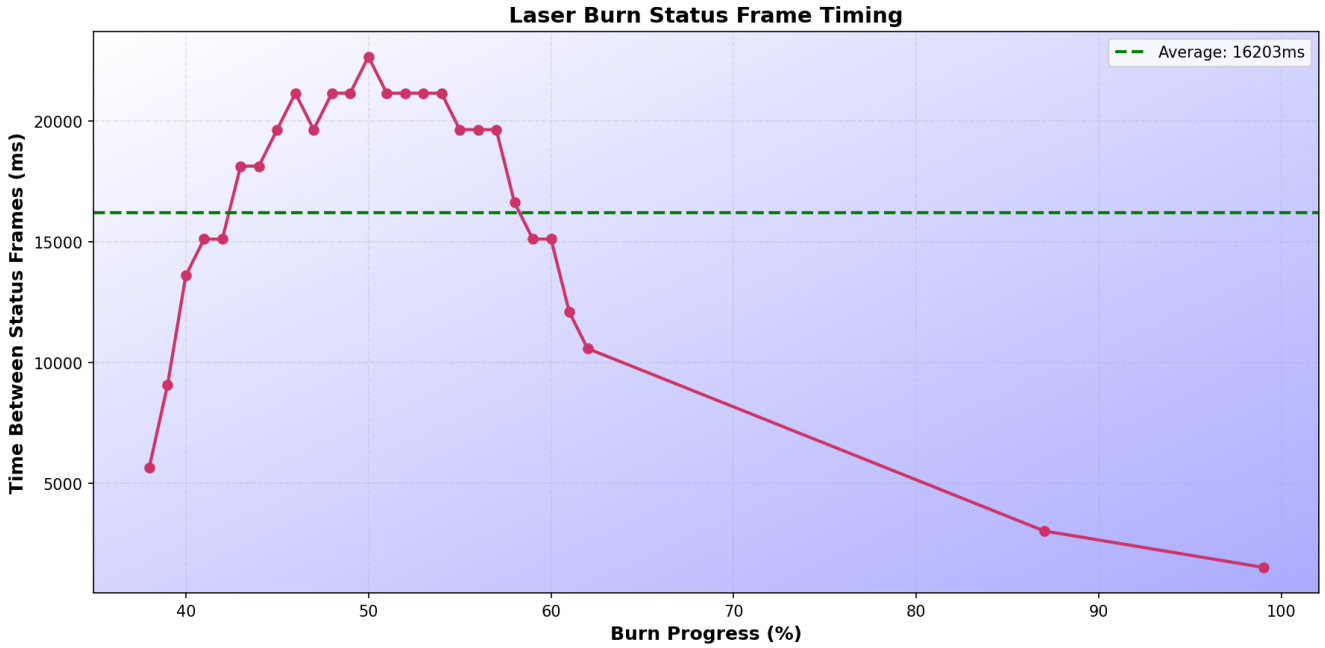


Figure 29. Burning a circle (20mm diameter)

When burning the laser sends completion heartbeats. I have the feeling one can see the vector circle reflected in them.

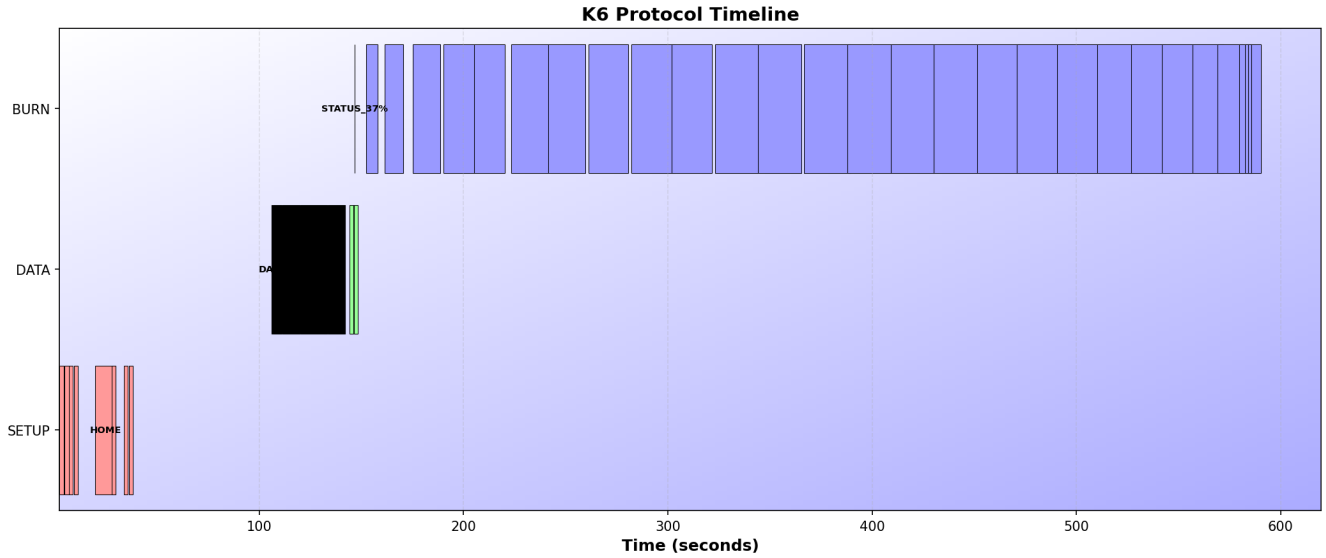


Figure 30. What the script spends it's time doing

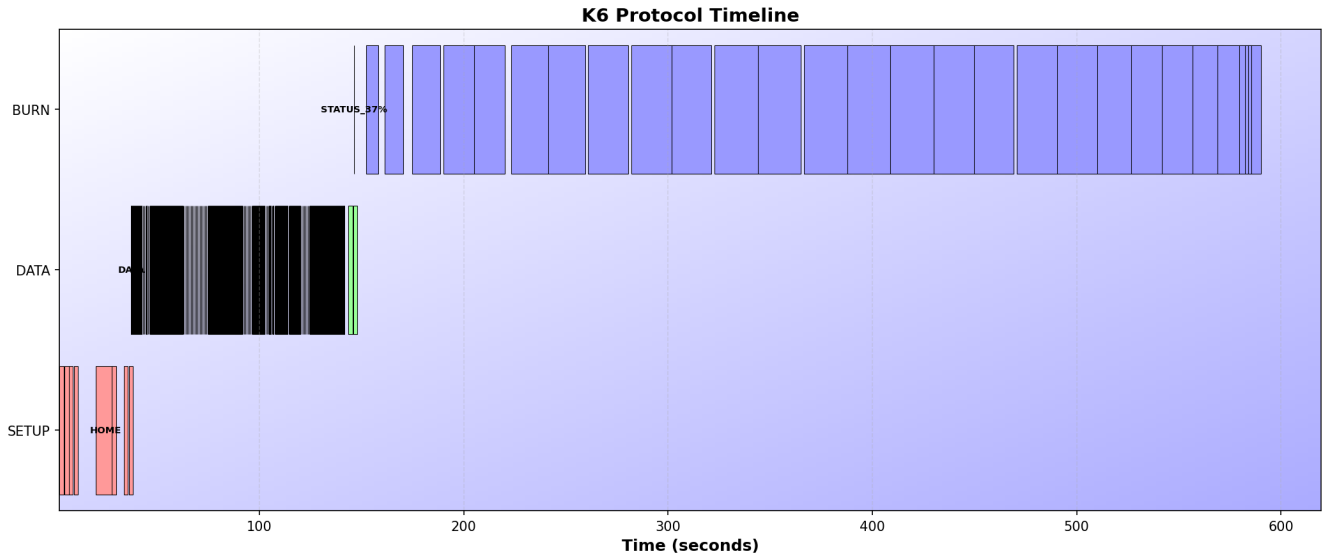


Figure 31. interleaving the data send and data prep work

The timing and the waiting for acks and hard coded timeouts etc. are adding a bit to the wait time but it seems to be acceptable now. The laser is taking the most time and so improving the timing is now not so large a win as it was at the start when conservative timing multiplied the time to image by at least 10 of what it is now.

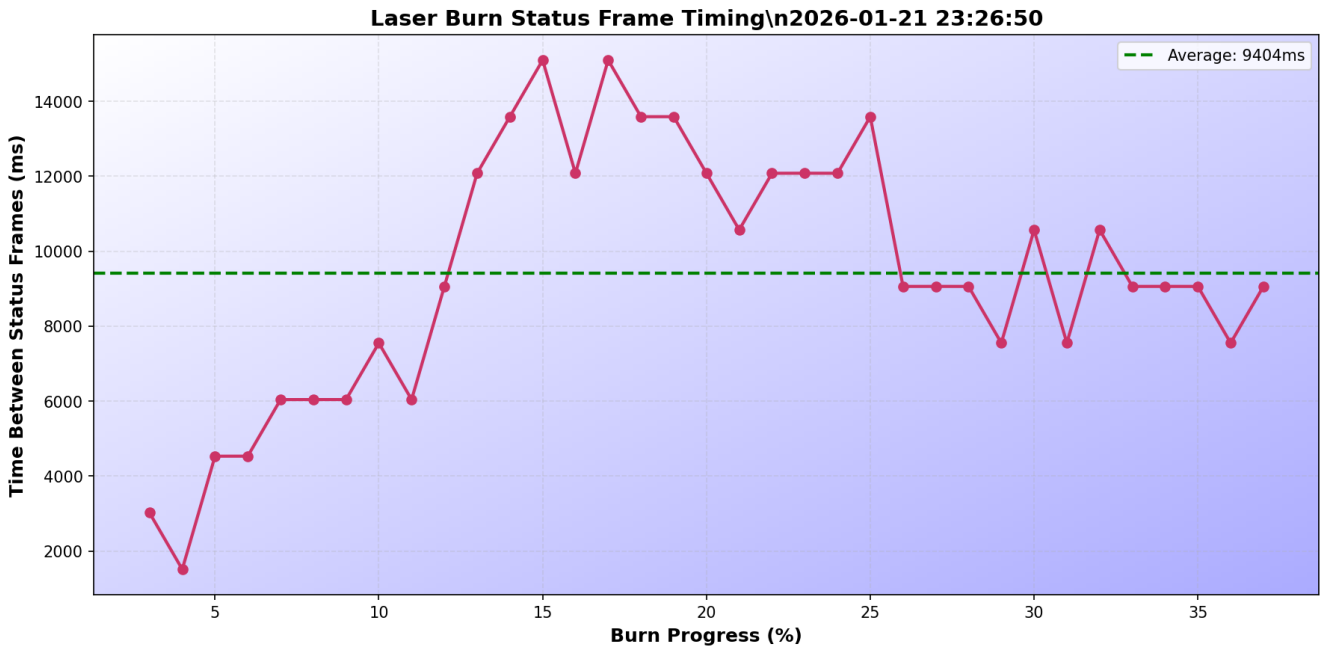


Figure 32. Burning a small tiger

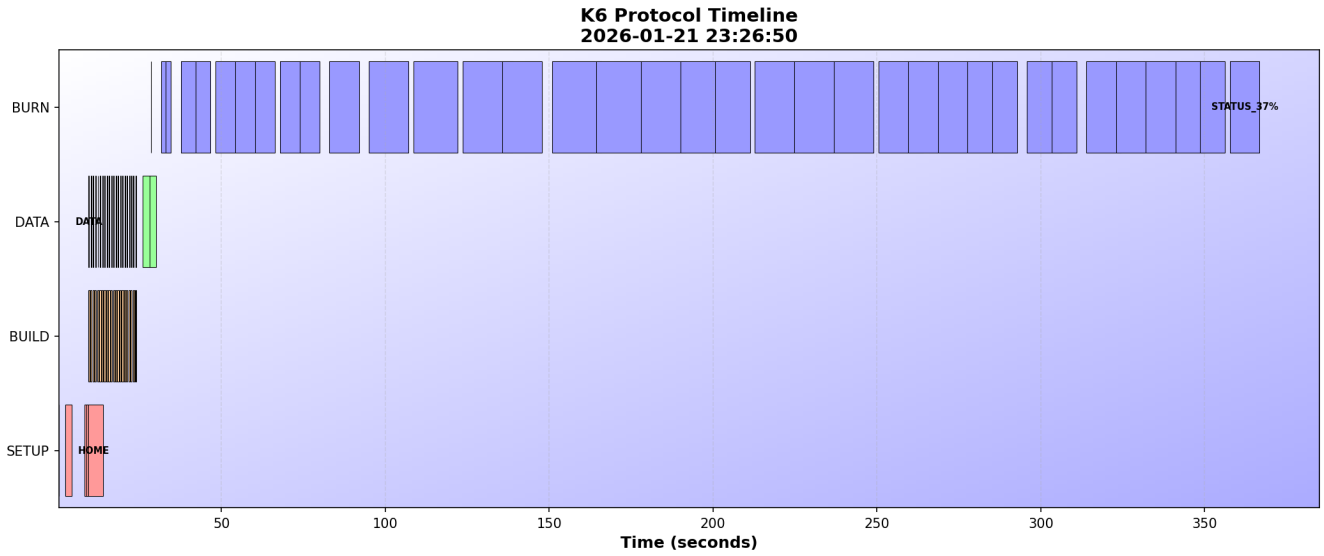


Figure 33. I was expecting it to take longer

Percentage completed is wonky. It was observed to start at 37% complete on a vector. It was also observed to end at 37% for a raster (but it looks like the laser did keep going after the script reported complete). We found the reason for the end at 37% and it's pure coincidence that one ends there and the other starts there.... see below. still need to look at why it starts at 37% complete on the vector and if that varies by vector size.

3.11.10.2. Burn Completion Detection Issues (2026-01-22)

Script quit at 37%. Laser kept burning.

Problems:

- `send_cmd()` left `ser.timeout=0.01s` after reading ACK
  - `wait_for_completion()` timed out every 10ms
  - Idle check failed after 30 seconds
- Estimated 336s. Burned 889s.
  - Hit max timeout at 37%
- Idle timeout: 30s. Device pauses 7-15s between status. Sometimes longer.

Fixed:

- Reset `ser.timeout=1.0` before monitoring
- Idle timeout: 90s (was 30s)
- Max timeout: 5× estimate (was 1×)
- Track exit reason
- Log everything to CSV

Exit Order:

1. See 100% → `COMPLETE_100%`

2. 90s silence → `IDLE_TIMEOUT`
3. Hit 5× estimate → `MAX_TIMEOUT`

Device Sends:

- `FF FF 00 XX` every 1.5s while burning
- Counts up by 1%
- May repeat final % or go silent

Tools Added:

- `serial_monitor.py` - watch serial live
- CSV - retry counts, states, exit reasons
- Error log - errors with tracebacks
- `--verbose` - full hex dumps

3.11.10.3. PlantUML Timing Diagrams

The CSV now includes `state` and `response_type` fields for generating detailed timing diagrams:

```
./generate_timing_diagram.py stat-26-01-21-23-26.csv -o timing.puml
plantuml timing.puml  # generates timing.png
```

Timing diagrams show:

- **Phase**: SETUP, BUILD, DATA, BURN
- **Serial TX/RX**: Binary high/low for transmission activity
- **Device Response**: ACK, HEARTBEAT, STATUS, TIMEOUT states
- **Burn Progress**: Percentage completion from device

This allows analysis of protocol timing, serial communication patterns, and device state transitions. The diagrams are of course a mile wide so to be viewed when needed and then scrolled. They are not suitable for documentation other than as cropped versions.

The timing diagram script is a work in progress.

Known issues

- some overlap

Appendix A: Architecture Decision Records

All ADRs for the Wainlux K6 project.

A.1. Format

Each ADR follows standard structure

- Status (Proposed/Accepted/Deprecated/Superseded)
- Context (the problem)
- Decision (what we chose)
- Rationale (why)
- Consequences (trade-offs)

A.2. Records

A.2.1. ADR-001: Base Image Selection

A.2.1.1. Status

Accepted

A.2.1.2. Context

Pi Zero W is ARMv6. **Not a choice - it's what we have.** Hardware constraint, not decision.

Given Pi Zero W, need Docker base image.

Options considered

- Alpine Linux
- Official Raspberry Pi images (raspbian/bullseye)
- Balena IoT images
- Debian slim
- Ubuntu (if ARMv7+ available)

A.2.1.3. Decision

Use `balenalib/raspberry-pi:bullseye`.

A.2.1.4. Rationale

ARMv6 support: Pi Zero W requirement. Alpine dropped ARMv6 in 3.13+. Ubuntu doesn't support ARMv6.

Size: 120MB vs 200MB+ official Raspberry Pi images. Debian slim 180MB.

Tested: Balena production-tested for IoT. Official images stable but heavier.

Build time: Pre-built layers save hours on first build vs building from scratch.

Maintenance: Balena actively maintains ARMv6 images for IoT fleet.

Considered Alpine (smallest) but ARMv6 support dropped. Would need to pin old version (risky).

Official Raspberry Pi images work fine but 80MB larger for minimal gain.

Debian slim is middle ground but not IoT-optimized.

A.2.1.5. Consequences

Positive

- Stable ARMv6 platform
- Smaller image = faster builds
- IoT-optimized base
- Active maintenance

Negative

- Balena-specific (less common than Alpine/Debian)
- Locked to Balena update schedule

Trade-off accepted: ARMv6 support and size win outweigh platform specificity.

A.2.2. ADR-002: Serial vs USB Communication

A.2.2.1. Status

Accepted

A.2.2.2. Context

K6 uses CP2102 USB-to-serial bridge chip.

Options considered

- pyserial (serial port abstraction)
- pyusb (direct USB access)
- libusb1 (C bindings)
- Custom USB driver

A.2.2.3. Decision

Use pyserial with /dev/ttyUSB0.

A.2.2.4. Rationale

CP2102 presents as serial port. Linux kernel driver built-in.

pyserial: * Standard serial API * Abstraction over platform differences * No USB protocol knowledge required * Device appears as file * Works with existing protocol code

pyusb considered: * Direct USB control * More complex * Requires USB descriptor knowledge * No

advantage for serial device

libusb1: * C library, Python bindings clunky * Lower level than needed * More code, more bugs

Custom driver: * Massive overkill * Kernel module complexity * CP2102 driver already exists

Binary protocol runs cleanly over serial read/write operations.

A.2.2.5. Consequences

Positive: * Simple serial read/write operations * No USB complexity or libusb dependencies * Standard Linux device permissions * Portable code (works on any Linux)

Negative: * No access to low-level USB features (not needed) * Depends on kernel driver (already present)

Trade-off accepted: Simplicity and portability win. No need for USB-level control.

A.2.3. ADR-003: Flask vs FastAPI

A.2.3.1. Status

Accepted

A.2.3.2. Context

Need web framework for Pi Zero W (512MB RAM).

Options

- Flask
- FastAPI
- Django

A.2.3.3. Decision

Use Flask.

A.2.3.4. Rationale

Memory: Flask uses 40MB less RAM than FastAPI.

Async: Don’t need async. Serial communication is synchronous. Single laser. One job at a time.

Simplicity: Flask proven on Pi Zero. Minimal dependencies.

Speed: FastAPI async overhead wasted on serial I/O.

Django too heavy for this use case.

A.2.3.5. Consequences

Lower memory footprint on constrained hardware.

Simpler code without async complexity.

Single-threaded model matches serial hardware constraints.

A.2.4. ADR-004: No Database

A.2.4.1. Status

Accepted

A.2.4.2. Context

Web app needs state management. Could use database.

A.2.4.3. Decision

Stateless design. No database.

A.2.4.4. Rationale

RAM: 512MB total. Database uses 50-100MB.

Purpose: Real-time laser control, not job management.

Complexity: Database adds failure modes.

State: Restart clears state. This is acceptable for single-user local control.

Jobs run immediately. No queue. No persistence needed.

A.2.4.5. Consequences

Positive: * Lower memory usage * Simpler deployment * Fewer dependencies * Faster restarts

Negative: * No job history * No job queue * Must re-upload after restart

A.2.4.6. Mitigation

CSV logs provide burn history.

User workflow is immediate: upload → burn → done.

For job queues, add database later if needed.

A.2.5. ADR-005: Bash Scripts for Deployment

A.2.5.1. Status

Accepted

A.2.5.2. Context

Single Pi Zero W. One Docker container. Need deployment automation? Furher background.: IF someone else wants to use this simplicity will be better.

Options considered:

- OpenTofu/Terraform
- Ansible
- Balena
- Bash scripts
- Make

A.2.5.3. Decision

Use bash scripts.

A.2.5.4. Rationale

A.2.5.4.1. Requirements

- Deploy code to one device
- keep it standalone
- Build Docker image
- Start container
- View logs
- Low complexity

A.2.5.4.2. Why Bash

Simple. Direct. Fast.

50 lines does the job. No dependencies. Easy to debug. Runs anywhere.

OpenTofu adds 500+ lines. Needs state management. Overkill for one device.

Ansible better than Tofu. Still too much for one device.

Balena for fleets. We have one Pi.

Make just wraps bash. No gain.

A.2.5.4.3. Trade-offs

Pros:

- Zero dependencies
- Fast execution
- Easy debugging
- Self-contained
- Works in Flatpak
- Clear error messages
- No state files
- No abstractions

Cons:

- Manual for multiple devices
- No drift detection
- No declarative model
- Harder to test

A.2.5.4.4. When to Reconsider

Switch to Ansible at 3+ devices.

Never use Terraform/OpenTofu for this.

A.2.5.5. Consequences

A.2.5.5.1. Positive

- Deployment works now
- Anyone can read the script
- Easy to modify
- No new tools to learn
- Follows KISS principle

A.2.5.5.2. Negative

- Manual sync for multi-device
- Script must handle errors
- No automatic rollback

A.2.5.5.3. Mitigation

- Add error handling to scripts

- Keep scripts short
- Document commands
- Use systemd for auto-restart

A.2.5.6. Implementation

Scripts in `scripts/`:

- `deploy_to_pi.sh` - Full deployment
- `sync_with_pi.sh` - Bidirectional code sync (push/pull)
- `quick_deploy.sh` - Skip sync

All use standard tools: `ssh`, `scp`, `tar`, `docker`.

A.2.5.7. Notes

This is the right tool for the job. Simple problem. Simple solution.

IaC tools solve different problems: multi-region cloud deployments, hundreds of resources, team coordination.

We have: one Pi, one container, one person.

Bash wins.

A.2.6. ADR-006: Clean Room Reverse Engineering

A.2.6.1. Status

Accepted

A.2.6.2. Context

K6 protocol undocumented. Initial assumption: existing K3 open source code would work.

Assumption failed: K3 protocol differs from K6. K3 handshake stalled. K3 opcodes wrong.

Need interoperability without vendor documentation.

Options considered

- Use existing K3 OSS implementation as-is (tried, failed)
- Adapt K3 code with modifications (insufficient)
- Use vendor SDK (can't find one)
- Clean room reverse engineering
- Black box only (no verification)
- Contact vendor for documentation (no response expected)

A.2.6.3. Decision

Clean room reverse engineering with verification via decompilation.

A.2.6.4. Rationale

A.2.6.4.1. Method

1. USB packet capture
2. Serial monitoring (115200 baud)
3. Black box testing
4. Public K3 protocol references
5. Decompilation for **verification only**

A.2.6.4.2. Critical Distinction

Decompiled code used for verification. Not implementation.

Process:

1. Observe protocol via USB captures
2. Implement from observed packets
3. Verify with Ghidra decompilation
4. Test on hardware

Not used from vendor code:

- Algorithms
- Variable names
- Code structure
- Direct translations

A.2.6.4.3. AI-Mediated Analysis

Ghidra accessed via MCP with Claude as intermediary.

LLM answered protocol questions without user reading source line-by-line.

Creates separation between implementation and vendor internals.

A.2.6.4.4. Legal Basis

Decompilation for interoperability is fair use under US precedent:

- *Sega Enterprises Ltd. v. Accolade, Inc.*, 977 F.2d 1510 (9th Cir. 1992)
    - Held: Intermediate copying for purpose of understanding functional requirements for interoperability is fair use

    - Public sources: Wikipedia
- *Sony Computer Entertainment, Inc. v. Connectix Corp.*, 203 F.3d 596 (9th Cir. 2000)
    - Held: Reverse engineering of software for interoperability purposes is fair use
    - Public sources: Wikipedia

Both cases establish that reverse engineering for interoperability purposes constitutes fair use.

No vendor code included in repository.

All distributed code is original implementation.

A.2.6.5. Consequences

A.2.6.5.1. Positive

- Legal distribution under MIT/CC
- No copyright infringement
- No trade secret issues
- Community can safely build on this work
- Protocol validated by actual device behavior

A.2.6.5.2. Negative

- More work than using vendor SDK
- Some protocol details require testing to confirm
- Documentation responsibility on us

A.2.6.5.3. Verification

Protocol correctness validated by:

- Device ACK/NAK responses
- Actual laser movement and marking
- USB traffic comparison
- Hardware testing

A.2.6.6. Implementation

See ../CLEAN_ROOM.md for full methodology and contributor guidelines.

All code in `/docker-wainlux` and `/scripts` is original work based on observed protocol behavior.

A.2.7. ADR-007: Documentation Format

A.2.7.1. Status

Accepted

A.2.7.2. Context

Need documentation that works for

- GitHub viewing
- PDF generation
- Multi-format output
- Technical diagrams

Options

- Markdown
- AsciiDoc
- reStructuredText
- LaTeX
- Plain text (.txt)
- HTML/Wiki
- Word (.docx)

A.2.7.3. Decision

Use AsciiDoc.

A.2.7.4. Rationale

GitHub rendering: Native support. Clean display.

Multi-format: Single source → HTML, PDF, EPUB.

Includes: Can split large docs into modules. Compose via `include::`.

Diagrams: PlantUML integration.

Book publishing: Proper chapters, TOC, cross-refs.

Technical writing: Callouts, admonitions, source blocks with syntax.

Alternatives rejected:

- **Markdown:** Too limited. No includes. Poor PDF output.
- **LaTeX:** Overkill. Hard to read plain text.
- **reStructuredText:** Python-centric. Less common.
- **Plain text:** No formatting. No diagrams. No output formats.
- **HTML/Wiki:** Hard to version control. Poor plain text readability. Not portable.

- **Word (.docx):** Proprietary. Microsoft format. Poor Git diffs. Best kept in SharePoint, not Git repos.

A.2.7.5. Consequences

A.2.7.5.1. Positive

- Single source for multiple formats
- Clean GitHub rendering
- Professional PDF output
- Modular documentation structure
- PlantUML diagrams in-line (possible but not used)

A.2.7.5.2. Negative

- Less common than Markdown
- Requires asciidoctor for PDF
- Steeper learning curve

A.2.7.5.3. Convention

- Main docs: `.asciidoc`
- Includes: `.adoc`
- Images: PlantUML in `images/` dir
- Formatting: `include-formatting-book-header.adoc`

A.2.7.6. Implementation

Files

- `README.asciidoc` - Entry point
- `documentation/*.adoc` - Modular sections
- `images/*.plantuml` - Diagrams
- `ADR/*.adoc` - This file and others

Build

- GitHub renders automatically
- Local: `asciidoctor README.asciidoc`
- PDF: `asciidoctor-pdf README.asciidoc`

A.2.8. ADR-008: Ghidra via MCP for Analysis

A.2.8.1. Status

Accepted

A.2.8.2. Context

Need to verify protocol understanding without directly reading vendor code.

Options

- Manual Ghidra analysis (direct code reading)
- IDA Pro decompilation
- Ghidra via MCP with LLM intermediary
- Skip verification entirely

A.2.8.3. Decision

Use Ghidra via Model Context Protocol (MCP) with Claude as intermediary.

A.2.8.4. Rationale

Separation: LLM reads decompiled code. User gets structured answers.

Specific queries: "What's the max value for depth parameter?" vs reading full source.

No direct exposure: User doesn't read vendor code line-by-line.

Verification only: Confirms observed behavior, doesn't drive implementation.

A.2.8.4.1. Clean Room Advantage

Traditional clean room: two teams, one reads source, one implements.

MCP approach: LLM reads, user implements. Similar separation.

User can't accidentally copy code patterns from unseen source.

A.2.8.4.2. Alternatives Rejected

Manual Ghidra: Too much direct code exposure.

Skip verification: Higher risk of protocol errors and potential to brick device.

IDA Pro: found ghidra mcp quicker.

A.2.8.5. Consequences

A.2.8.5.1. Positive

- Verify protocol understanding
- Reduce implementation errors
- Maintain clean room separation
- Answer specific questions quickly
- Legal defensibility

A.2.8.5.2. Negative

- Not strict two-team clean room
- Requires MCP server setup
- LLM costs for queries
- User could read files directly if desired

A.2.8.5.3. Protocol

1. Observe protocol via USB/serial
2. Implement from observations
3. Query MCP: "Does decompiled code show depth range 1-255?"
4. LLM answers from Ghidra output
5. User validates answer against hardware
6. Adjust implementation if needed

A.2.8.6. Implementation

See ../documentation/option-ghidra-mcp-setup.adoc for setup.

MCP server: pyghidra-mcp

Used during protocol development. Not required for running the software.

A.2.9. ADR-009: Docker Containerization

A.2.9.1. Status

Accepted

A.2.9.2. Context

Need to deploy Flask app to Pi Zero W. Must handle Python dependencies (Pillow, pyserial, Flask).

Options

- Docker containerization
- Bare Pi Python installation (manual pip install)
- Virtual environment with systemd service
- Package as .deb

A.2.9.3. Decision

Use Docker containerization.

A.2.9.4. Rationale

Recipe-based deployment: Dockerfile is executable documentation. No hidden steps. No "it works on my machine."

Migration path: Container runs anywhere. Move to different Pi? Copy container. Move to x86? Rebuild from same Dockerfile.

Isolation: System dependencies (libjpeg, libxcb) in container. No system pollution. Clean uninstall = remove container.

Reproducibility: Same Dockerfile → same environment. Every build identical.

Version control: Infrastructure as code. Dockerfile in git. Changes tracked.

Considered bare Python: Faster boot, simpler. But deployment = manual documentation. Migration = rewrite install steps.

Virtual environment: Better than bare but still system-dependent. Migration still requires docs.

A.2.9.5. Consequences

Positive

- Deployment = recipe, not procedure
- Migration to new hardware trivial
- Dependencies isolated
- Build reproducible
- Easy rollback (image tags)

Negative

- Docker layer adds complexity
- Slightly slower startup
- Need Docker knowledge
- More disk space (120MB base image)

Trade-off accepted: Portability and reproducibility outweigh Docker overhead.

A.2.10. ADR-010: Bytefield Protocol Diagrams

A.2.10.1. Status

Accepted

A.2.10.2. Context

K6 protocol has binary packets: opcodes, sizes, depths, powers, pixel data. Need RFC-style diagrams showing byte layout proportionally.

Current state: Manual ASCII art in AsciiDoc. Not sourced from structured data. Duplication between docs and code.

Goal: Structured packet spec → visual diagram. Readable spec. Easy visual check.

Options

- Protocol (Python CLI) - ASCII from command-line string
- Bytefield-SVG (npm) - SVG from Clojure DSL or JSON
- PacketDiag (Python) - PNG/SVG from text DSL
- Ditaa (Java) - PNG from ASCII art
- PlantUML Salt - PNG/SVG from PlantUML syntax
- LaTeX bytefield - PDF/PNG from LaTeX
- Manual ASCII in AsciiDoc - What we have now

A.2.10.3. Decision

Use **Bytefield-SVG** with Docker container.

Documentation: https://bytefield-svg.deepsymmetry.org/bytefield-svg/1.11.0/intro.html

A.2.10.4. Rationale

SVG output: Scales perfectly. Professional appearance. GitHub renders inline.

JSON input: Version controlled packet specs. Single source of truth. Readable by humans and tools.

Podman container: No Node.js on Pi required. Build once, run anywhere. Consistent environment. Podman = Docker-compatible, rootless, Flatpak-friendly via `host-spawn`.

Active maintenance: Bytefield-SVG actively developed. Well-documented. Good examples.

Workflow: Write JSON spec → run container → generate SVG → commit both. Spec in git, diagram regenerated on change.

Considered keeping ASCII but SVG quality justifies container complexity. Container is lightweight (Node.js Alpine ~50MB).

Alternative (PacketDiag) rejected: Python 2 legacy, DSL not JSON, uncertain maintenance.

Alternative (Protocol CLI) rejected: ASCII only, command-line specs not version controlled.

A.2.10.5. Implementation

Docker container: `docker-bytefield/`

Generate script: `scripts/generate_diagrams.sh`

Packet specs: `images/*.json`

Output diagrams: `images/*.svg`

Container build:

```
cd docker-bytefield
host-spawn podman build -t bytefield-svg .
```

Generate diagrams:

```
scripts/generate_diagrams.sh
```

Or direct:

```
host-spawn podman run --rm -v $(pwd)/images:/diagrams:Z bytefield-svg input.json -o
output.svg
```

Flatpak users: Uses `host-spawn podman` to access host system's podman. No extensions needed.

A.2.10.6. Consequences

A.2.10.6.1. Protocol (Python CLI)

Input: Command-line string `"Field:bits,Field2:bits"`

Output: ASCII art

Pros: - Simple syntax - Python (in stack) - Works immediately - Good for ad-hoc diagrams

Cons: - No structured input file - ASCII only (no proportional rendering) - Last updated 7 years ago - Spec not version controlled separately

Example:

```
protocol "Opcode:8,Size:16,Depth:8,Power:16,Data:320"
```

A.2.10.6.2. Bytefield-SVG (npm)

Input: Clojure DSL or JSON

Output: SVG

Pros: - Beautiful proportional output - Active maintenance - JSON input = version control - Scales perfectly - Web + CLI

Cons: - Requires Node.js (not in Pi stack) - Clojure DSL learning curve - JSON schema not standardized - Build step required

Example JSON:

```
{
  "fields": [
    {"name": "Opcode", "bits": 8},
    {"name": "Size", "bits": 16}
  ]
}
```

Link: https://github.com/Deep-Symmetry/bytefield-svg

A.2.10.6.3. PacketDiag (Blockdiag)

Input: Simple text DSL

Output: PNG/SVG/PDF

Pros: - Clean readable syntax - Python-based - Part of blockdiag suite - Proportional output

Cons: - Python 2 legacy - Maintenance unclear - Less common than Protocol

Example:

```
packetdiag {
  colwidth = 32
  0-7: Opcode
  8-23: Size
  24-31: Depth
}
```

A.2.10.6.4. Ditaa (ASCII to Diagram)

Input: ASCII art

Output: PNG

Pros:

- PlantUML ecosystem (already using)
- Manual control
- AsciiDoc built-in support

Cons:

- Manual drawing
- Not proportional by default
- ASCII maintenance overhead

A.2.10.6.5. PlantUML Salt

Input: PlantUML salt syntax

Output: PNG/SVG

Pros:

- Already in toolchain
- Existing `plantuml::` support

Cons:

- Not designed for packet diagrams
- Limited proportional sizing
- Awkward for protocol work

A.2.10.6.6. LaTeX bytefield

Input: LaTeX markup

Output: PDF/PNG

Pros:

- *Publication quality
- Precise control
- Standard in academia

Cons:

- LaTeX dependency
- Complex build pipeline
- Overkill for Pi project

A.2.10.6.7. Manual ASCII (Current)

Input:

- Hand-written ASCII in AsciiDoc

Output:

- Text in docs

Pros:

- Zero dependencies
- Works now
- Full control
- GitHub renders directly

Cons:

- No structured source
- Duplication (spec in code, diagram in docs)
- Manual alignment
- Not proportional
- Easy to get wrong

A.2.10.7. Evaluation Criteria

Must have:

- Structured input (JSON/YAML/DSL)
- Version controlled
- Readable by humans
- Generates visual output

Nice to have:

- Python-based (Pi stack)
- Proportional rendering
- Active maintenance
- Single source of truth

Trade-offs:

- ASCII = simple, ugly
- SVG = beautiful, complex
- Python = in stack
- Node.js = not in stack

A.2.10.8. Proposed Path

Phase 1 (now):

- Keep manual ASCII. Document packets in JSON separately.

Phase 2 (evaluate):

- Install Node.js. Test Bytefield-SVG. Generate diagrams from JSON.

Phase 3 (decide):

- If Bytefield-SVG works → adopt. If not → PacketDiag or keep ASCII.

A.2.10.9. Open Questions

- Worth Node.js dependency for SVG output?
- JSON schema: own format vs Bytefield-SVG format?
- Generate on build vs commit generated images?

- Single JSON with all packets vs one per packet?

A.2.10.10. Consequences

Positive

- Beautiful proportional SVG diagrams
- JSON specs = single source of truth
- Version controlled packet definitions
- No Node.js dependency on Pi (containerized)
- Regenerate diagrams from specs anytime
- GitHub renders SVG inline

Negative

- Docker build step required
- Slightly more complex than manual ASCII
- Must maintain JSON schema
- Generated files in git (or regenerate on build)

Trade-off accepted:

- Diagram quality and structured specs outweigh Docker complexity.

A.2.10.11. Alternatives Considered

See full analysis above. Key rejections:

- **Protocol (Python CLI):** ASCII only, no structured source
- **PacketDiag:** Python 2 legacy, DSL not JSON
- **Manual ASCII:** Works but no structured source, duplication
- **LaTeX bytefield:** Overkill, complex build
- **PlantUML Salt:** Not designed for protocol work

Bytefield-SVG chosen for: best output quality, JSON input, active maintenance, Docker solves Node.js dependency.

A.2.10.12. Example: K6 Raster Packet Diagram

K6 Raster Data Packet (0x22) Specification

```
;; K6 Raster Data Packet (0x22)
;; 9-byte header + variable pixel data

;; Color coding
(defattrs :bg-header {:fill "#e8f4f8"})
(defattrs :bg-data {:fill "#fff4e8"})

(draw-column-headers)
```

```
;; Header row 1
(draw-box "0x22 (Raster Data)" [:box-first {:span 8} :bg-header])
(draw-box "Size MSB (16-bit BE)" [:box-related {:span 8} :bg-header])
(draw-box "Size LSB" [:box-related {:span 8} :bg-header])
(draw-box "Depth (1-255)" [:box-last {:span 8} :bg-header])

;; Header row 2
(next-row)
(draw-box "Power MSB (16-bit BE, 0-1000)" [:box-first {:span 8} :bg-header])
(draw-box "Power LSB" [:box-related {:span 8} :bg-header])
(draw-box "Line MSB (16-bit BE)" [:box-related {:span 8} :bg-header])
(draw-box "Line LSB" [:box-last {:span 8} :bg-header])

;; Header row 3 and pixel data
(next-row)
(draw-box "Pixel Count" [{:span 8} :bg-header])
(draw-gap "Packed Pixel Data (1-bit packed)" [{:span 24} :bg-data])

;; Variable length continuation
(next-row)
(draw-gap "..." [{:span 32} :bg-data])
```

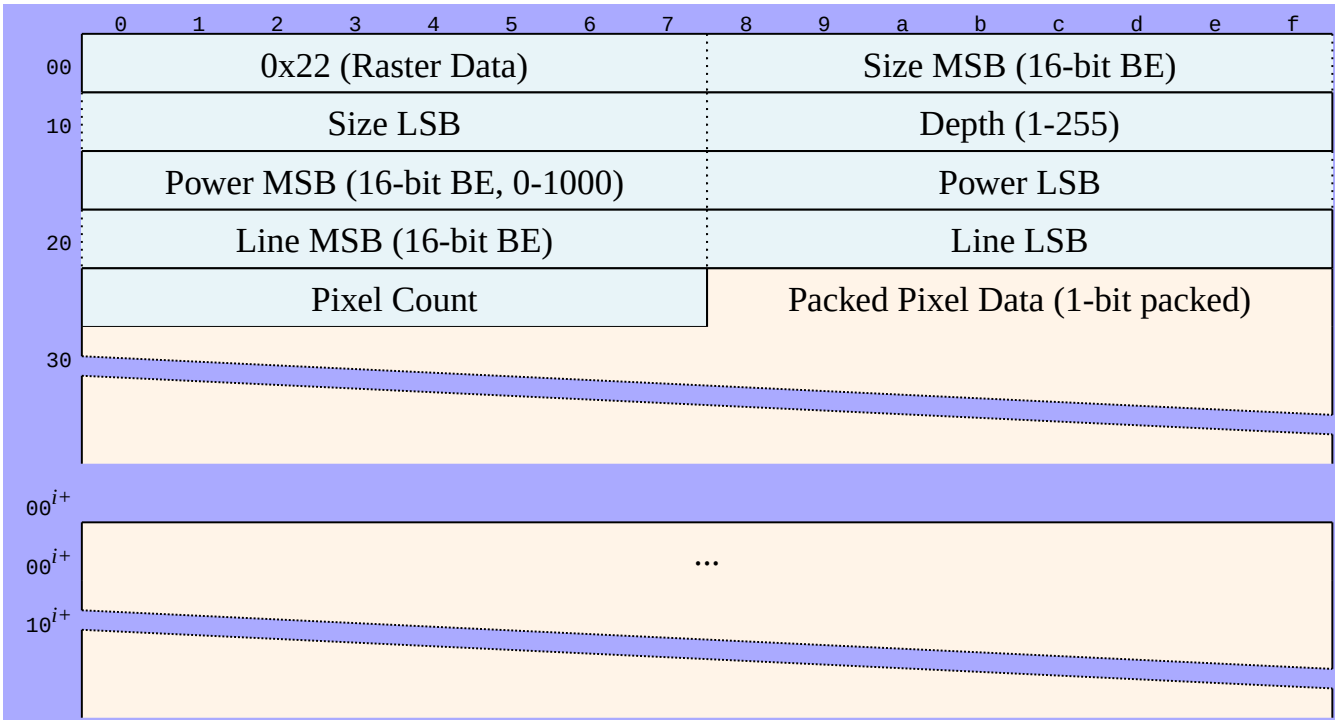


Figure 34. Generated K6 Raster Packet Diagram

Appendix B: Documentation Structure

B.1. AI executive summary

This document explains the project's documentation structure using AsciiDoc. It starts with why

AsciiDoc over Markdown (flexibility for multiple formats, GitHub rendering), what AsciiDoc offers (includes, multi-format output, editing tools), and how to structure docs by splitting into main .asciidoc and included .adoc files with conditional formatting via flag-book. It covers directory handling (imagesdir, localdir) with save/restore patterns for standalone vs. included viewing, including code examples for setup (directly under title headings) and restore (at file end). The goal is coherent, near-code docs following standards like TOGAF DevOps guides. All statements are verified as true or defensible.

B.2. Why AsciiDoc?

Why not Markdown? It is the default for READMEs. To default means to fail. If no other is there then use this is what it means. it means we've failed if we use MD. My personal interpretation that is true for github.

Markdown is GitHub's default for READMEs. AsciiDoc is superior for complex docs. GitHub renders .adoc if present.

I prefer AsciiDoc for its ability to convert to PDF, standalone HTML, or other formats. GitHub displays .adoc over .md if both exist.

Asciidoctor supports multiple outputs. GitHub prioritizes .adoc over .md.

AsciiDoc supports structured documents with includes, images, and links.

B.3. What is AsciiDoc?

AsciiDoc allows includes like `include::filename[leveloffset=+2]`. Split documents into multiple files for easier editing near code, following standards like TOGAF for DevOps.

TOGAF has DevOps guides, such as "Using TOGAF to Define and Govern a DevOps Environment," recommending docs near code.

It supports images, links, and parsing into formats via Asciidoctor and Asciidoctor-PDF. Even DOCX if needed.

Edit in VS Code with preview. Parse in GitHub Actions for Pages.

Extensions like AsciiDoc available. Actions can use asciidoctor.

Add templates for styling. Focus on content, less mouse use.

In the end the distributed docs become one coherent document.

B.4. Structure

Split into files. Main file as .asciidoc, includes as .adoc.

Set `:flag-book: true` in main doc.

Use `ifdef` and `ifndef` in includes for standalone viewing with proper formatting.

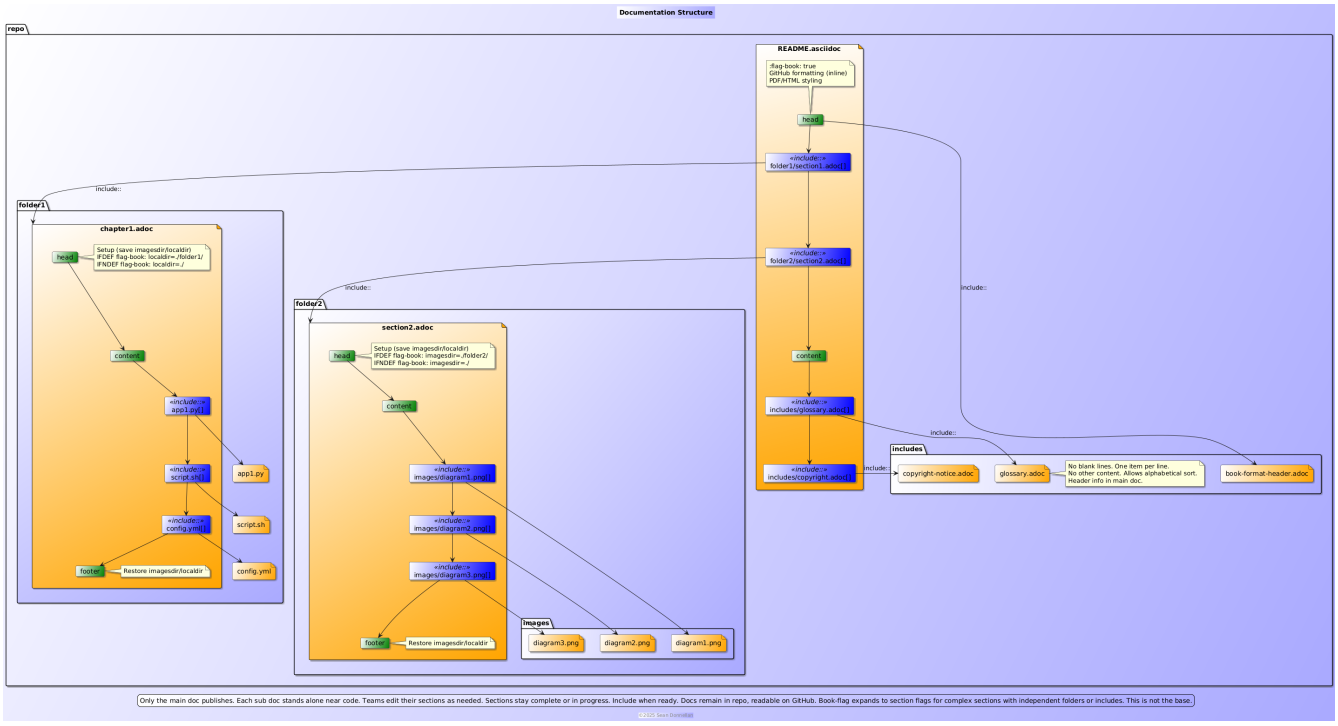


Figure 35. Overview of doc structure logic

B.5. Handling Directories in Includes

Save and restore `imagesdir` and `localdir` in each .adoc.

Set based on `flag-book` and `github-env`.

B.5.1. Setup for included docs (chapters, sections)

At top of included .adoc file

```
// Directly under = title heading, no blank lines
ifdef::flag-book[]
// Save variables and adjust for building from main doc
// Save current
:imagesdir-saved: {imagesdir}
:localdir-saved: {localdir}
// set in context from main doc
:localdir: ./subdir
:imagesdir: ./subdir/images
endif::flag-book[]
ifndef::flag-book[]
//add standalone formatting here
:toc: right
:toclevels: 5
:sectnums:
:sectnumlevels: 5
// GitHub emoji icons for admonition blocks
ifdef::github-env[]
:icons: font
:tip-caption: :bulb:
```

```
:note-caption: :information_source:
:important-caption: :heavy_exclamation_mark:
:caution-caption: :fire:
:warning-caption: :warning:
endif::github-env[]
ifdef::github-env[]
//Add github formatting like icons and stuff here
endif::github-env[]
endif::flag-book[]
// leave a blank line here:

// Content starts here
....
```

B.5.2. Restore at End

At end of included .adoc file

```
// END of File
// leave a blank line here:

ifdef::flag-book[]
// At end of file
// Restore
:imagesdir: {imagesdir-saved}
:localdir: {localdir-saved}
endif::flag-book[]
```

B.6. Main Document Setup

The main .asciidoc file sets `:flag-book: true` and includes sub-docs.

GitHub formatting must be inline, as GitHub does not load includes. Add GitHub-specific attributes or content directly in the main file.

HTML/PDF/etc styles can be as includes, since processing tools handle them.

B.6.1. Example Main doc Styling

Top of main .asciidoc file

```
// directly under = title heading, no blank lines
// Main document styling
:flag-book: true
//PDF styling
:pdf-theme: custom
:pdf-fontsdir: fonts/
:pdf-style: theme.yml
//Example HTML Styling
```

```
:stylesheet: custom.css
:linkcss:
// Example GitHub Formatting with emoji icons for admonition blocks
ifdef::github-env[]
:icons: font
:tip-caption: :bulb:
:note-caption: :information_source:
:important-caption: :heavy_exclamation_mark:
:caution-caption: :fire:
:warning-caption: :warning:
endif::[]
// leave a blank line here:

// Content starts here
....
```

The emoji icon captions customize how NOTE, TIP, IMPORTANT, CAUTION, and WARNING admonition blocks render on GitHub. See https://github.com/jmriff/asciidoc for reference.

ACK

Acknowledgement byte (0x09) returned by K6 after successful command.

ARMv6

CPU architecture of Pi Zero W. 32-bit. Limited to older Alpine/Debian.

CP2102

USB-to-serial chip in K6. Creates /dev/ttyUSB0. Vendor 10c4:ea60.

Depth

Laser on-time per pixel. Range 1-255. Higher burns deeper.

Docker

Container platform. Isolates K6 app. Privileged mode for /dev access.

Flask

Python web framework. Lightweight. 40MB less RAM than FastAPI.

K3

Earlier Wainlux model. Protocol reverse-engineered. K6 differs.

K6

Wainlux K6 laser engraver. 80x80mm work area. USB serial. NOT GRBL.

Opcode

Command byte in protocol. 0x0A=connect, 0x17=home, 0x09=engrave (K3).

Pi Zero W

Raspberry Pi Zero W. ARMv6, 512MB RAM, WiFi. Headless host.

PySerial

Python serial library. Talks to /dev/ttyUSB0 at 115200 baud.

Raster

Image as rows of pixels. 8 pixels per byte packed.

TOGAF

The Open Group Architecture Framework. DevOps guides recommend docs near code.

Appendix C: Copyright and License

C.1. Code License

All source code in this repository is licensed under the **MIT License**.

See LICENSE for full text.

C.2. Protocol Documentation License

All protocol documentation (files in `/documentation`) is dual-licensed:

- **Creative Commons Attribution 4.0 International (CC BY 4.0)**, or
- **Creative Commons Zero v1.0 Universal (CC0 1.0)** - Public Domain Dedication

See LICENSE-DOCS for full text.

C.3. Clean Room Statement

This project was developed through clean-room reverse-engineering methods. No vendor source code, proprietary binaries, or confidential materials were used or redistributed.

See CLEAN_ROOM.md for full methodology and exclusions.

C.4. Copyright Notice