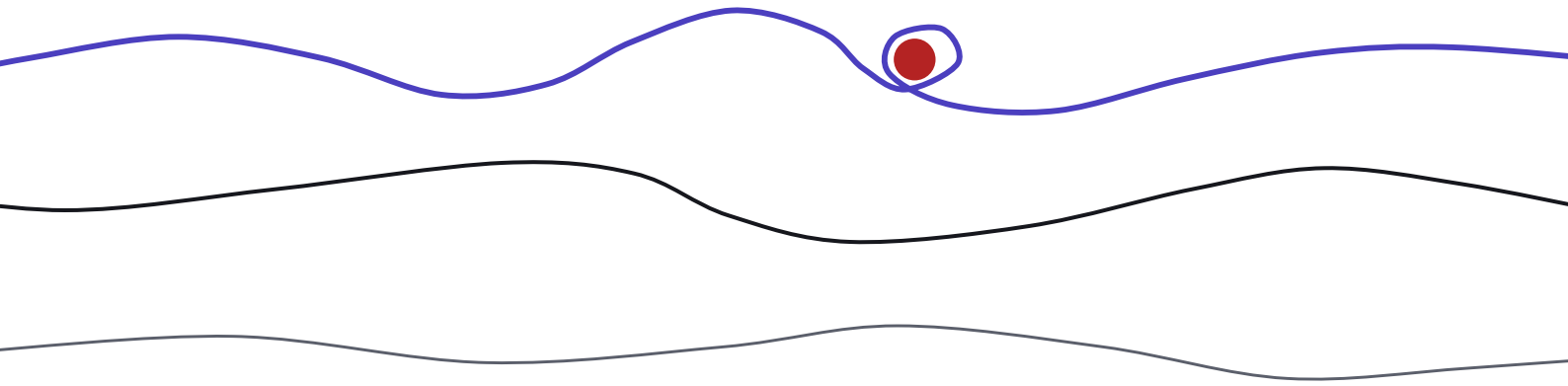
jupiter.au

Monochrome

ESP32-S3 firmware that drives a TTL monochrome MDA or Hercules type monitor at 720x350.



DATE

25 November 2026

AUTHOR

belikh

SOURCE

github.com/belikh/monochrome

Contents

Start

01 **Build and flash** 3

01.1 Build and flash 3

01.2 Check the sync before you connect video 3

Guides

02 **Send frames over Wi-Fi** 5

02.1 Set the network 5

02.2 Find the address 5

02.3 Send a test pattern 5

02.4 Send your own picture 6

02.5 Options 6

02.6 Choose a format 6

03 **Wire the monitor** 7

04 **Adjust the picture** 7

04.1 Centre the picture 7

04.2 Margin around the picture 8

05 **Choose a demo** 8

06 **Jupiter display demo suite** 8

06.1 Run the suite 9

06.2 Home Assistant and animated screens 9

06.3 Preview a screen 10

06.4 Display setup 10

Reference

07 **Pins** 11

07.1 Pixel values 11

08 **Timing** 12

09 **Frame protocol** 12

09.1 Pixel formats 12

09.2 TCP 13
09.3 UDP 13
09.4 One sender at a time 14
09.5 Measured speeds 14

10 **Glossary** 15

Concepts

11 **How the video works** 16
11.1 Frame buffers 16
11.2 Keep the bounce chunk count even 16
11.3 Reuse a frame buffer only after the driver latches the new one 17
11.4 Keep V-sync active low 17
11.5 Hercules 17

12 **How the 3D demo works** 18
12.1 The scene 18
12.2 Pixel output 18
12.3 Each frame 18
12.4 What made it fast 19

13 **Monochrome visual language** 19
13.1 Inherit from Jupiter 19
13.2 The display's visual grammar 20
13.3 Raster and layout rules 20
13.4 Shared source and generated outputs 21
13.5 Review checklist 21

Monochrome

Monochrome is ESP-IDF firmware for an ESP32-S3-DevKitC-1 (N16R8). It drives a TTL monochrome monitor of the *MDA* or Hercules type through a DE-9 connector. The firmware joins Wi-Fi and shows frames that a program on your network sends over TCP or UDP, with optional compression. It also includes a shaded 3D torus knot that runs at 49 frames per second, a wireframe screensaver and a calibration pattern, all drawn with three brightness levels.

Start with Build and flash, then Send frames over Wi-Fi.

SECTION

Start

Install it and run it for the first time.

01

Build and flash

Build the firmware with ESP-IDF 5.5 and flash it to an ESP32-S3-DevKitC-1.

You need ESP-IDF 5.5 and an ESP32-S3-DevKitC-1 with the N16R8 module (16 MB flash, 8 MB octal PSRAM).

01.1

Build and flash

1. Load the ESP-IDF environment in your shell.

   LOAD ESP-IDF

   ```
   . ~/esp/esp-idf/export.sh
   ```

2. Set the target and build.

   BUILD

   ```
   idf.py set-target esp32s3
   idf.py build
   ```

3. Plug in the board and flash it. Use the port that your system shows for the board.

   FLASH

   ```
   idf.py -p /dev/ttyACM0 flash monitor
   ```

The monitor prints `MDA video running: 720x350, pclk 16257000 Hz, mode 3` when the video driver has started. The default mode joins Wi-Fi and shows frames that you send. Set the network first. See Send frames over Wi-Fi. For a demo that needs no network, see Choose a demo.

> NOTE
>
> The DevKitC-1 has two USB ports. The port wired to the UART bridge carries the console log. Flashing worked through that port, and it showed up as `/dev/ttyACM0` on Linux.

01.2

Check the sync before you connect video

Wire the sync lines and ground first, as described in Wire the monitor. Leave the video wire unplugged and power the monitor. You should see a steady raster with no whine from the monitor. Connect the video wire once the monitor holds lock.

SEE ALSO

- Wire the monitor
- Choose a demo
- Adjust the picture

SECTION

Guides

Steps for one task each.

02

Send frames over Wi-Fi

Join the board to Wi-Fi, find its address and stream a picture to the monitor.

Mode 3 joins your Wi-Fi network and shows the frames that a program on your network sends. The board needs a 2.4 GHz network. The ESP32-S3 does not use 5 GHz.

02.1

Set the network

1. Open the configuration menu.

   CONFIGURE

   ```
   idf.py menuconfig
   ```

2. Open the Monochrome menu and enter the network name and password. Leave the password empty for an open network.
3. Build and flash. See Build and flash.

The settings are saved in `sdkconfig`, which Git ignores, so the password does not reach the repository.

02.2

Find the address

The monitor shows the board's IP address and port in large digits. The border is dim until the board has an address and bright once it has one. The same address appears in the log:

LOG

```
netfb: got IP 192.168.2.166, listening on port 5050
```

02.3

Send a test pattern

Run the sender from the repository on any computer on the same network. It needs only Python 3.

SCROLLING BARS OVER UDP

```
python3 tools/send.py 192.168.2.166 --demo --udp --fmt 2bpp --fps 50 --zlib
```

The monitor shows scrolling bars. Press Ctrl-C to stop. When a TCP client leaves or a UDP stream stops, the monitor keeps the last frame.

02.4

Send your own picture

With Pillow installed, send an image file. The sender scales it to 720x350.

SEND AN IMAGE

```
python3 tools/send.py 192.168.2.166 --image picture.png --fmt 1bpp --zlib
```

You can also send a raw frame file in any format. The file size must match the format. See Frame protocol.

SEND A RAW FRAME

```
python3 tools/send.py 192.168.2.166 --raw frame.bin --fmt 2bpp --udp --zlib
```

02.5

Options

OPTION	DEFAULT	MEANING
`--udp`	off	Send over UDP instead of TCP.
`--fmt`	`2bpp`	Pixel format: `1bpp`, `2bpp` or `8bpp`.
`--fps`	25	Frames per second to send.
`--zlib`	off	Compress frames (TCP) or rows (UDP).
`--delta`	off	UDP only: send only the rows that changed since the last frame.
`--port`	5050	The port that the board listens on.

02.6

Choose a format

- Use 1 bit per pixel for fast animation. It has no brightness levels.
- Use 2 bits per pixel for three levels. It needs about 2.5 MB/s uncompressed at 40 frames per second, which is near the limit of a typical Wi-Fi link. Use `--zlib` or stay at 30 frames per second or fewer.
- Use `--zlib` whenever the picture has flat areas, text or repeating patterns.
- Use `--delta` with UDP for a screen that mostly stays still.
- Avoid 8 bits per pixel over Wi-Fi unless you need the extra bits. The monitor can show only black, normal and bright, so 2 bits cover every level.

SEE ALSO

- Frame protocol
- Choose a demo

03

Wire the monitor

Connect four GPIOs and ground to the DE-9 connector of a TTL monochrome monitor.

Connect the board to the monitor's DE-9 connector as listed in Pins.

1. Connect both ground pins (1 and 2) of the connector to a board `GND` pin.
2. Connect GPIO 6 to pin 8 (H-sync) and GPIO 7 to pin 9 (V-sync).
3. Connect GPIO 4 to pin 7 (video).
4. Connect GPIO 5 to pin 6 (intensity), or leave pin 6 unconnected if you want one brightness level.

The GPIOs output 3.3 V and the monitor expects TTL levels. The setup this firmware was developed on drives the monitor directly from the GPIOs, with no buffer and no resistors, and the picture is stable. If your monitor misbehaves, put a 74HCT244 or 74AHCT125 buffer, or 150 ohm series resistors, between the GPIOs and the connector.

> WARNING
>
> Do not move the signals to GPIO 33 to 37 on an N16R8 board, because the octal PSRAM uses them. Avoid the strapping pins (0, 3, 45, 46) and the USB pins (19, 20).

SEE ALSO

- Pins
- Build and flash

04

Adjust the picture

Centre the image and understand why it can be smaller than the screen.

The picture position depends on the *Porch* values in `main/main.c`. The picture size depends on the monitor.

04.1

Centre the picture

To move the picture sideways, trade pixels between `hsync_front_porch` and `hsync_back_porch`. Keep their sum at 54. To move it vertically, trade lines between `vsync_front_porch` and `vsync_back_porch`. Keep their sum at 4.

SETTING	DEFAULT	SUM TO KEEP
`hsync_front_porch`	18	54 with `hsync_back_porch`
`hsync_back_porch`	36	54 with `hsync_front_porch`
`vsync_front_porch`	1	4 with `vsync_back_porch`
`vsync_back_porch`	3	4 with `vsync_front_porch`

Rebuild and flash after each change. See Build and flash. Set `MODE` to 0 to see the border and grid while you adjust. See Choose a demo.

04.2

Margin around the picture

On the INTRA monitor used to develop this firmware, a margin of about 2 cm remained around the grid. The signal is a standard 720x350 raster, and the monitor sets how much of the glass the picture covers. The monitor's front controls adjust brightness and contrast only.

05

Choose a demo

Switch between firmware graphics and the Jupiter green-phosphor demo suite.

The firmware has four modes. You choose one with the `MODE` macro in `main/main.c`.

MODE	SHOWS	FRAME BUFFERS
3 (default)	A picture that a program sends over Wi-Fi. Until one arrives, the monitor shows the board's IP address.	Two, in PSRAM
2	A shaded 3D torus knot with an orbiting light, at 49 fps	Two, in PSRAM
1	A wireframe icosahedron that tumbles and bounces around the screen	One, in internal RAM
0	A static calibration pattern: borders, grid, diagonals, intensity blocks, checkerboard and circle	One, in internal RAM

To change the mode:

1. Edit the line `#define MODE 3` in `main/main.c`.
2. Rebuild and flash. See Build and flash.

Mode 3 needs a Wi-Fi network. See Send frames over Wi-Fi. For Jupiter's adapted green-phosphor web demo screens, run `python3 tools/jupiter_display.py <board-ip> --demo all --fps 1 --repeat`. See Jupiter display demo suite.

Mode 2 prints its frame rate and the time spent in each stage every 100 frames, for example `49.3 fps, 17.0 ms render per frame, 813 of 1536 triangles drawn`.

Use mode 0 to centre the picture. See Adjust the picture.

SEE ALSO

- Send frames over Wi-Fi
- Jupiter display demo suite
- How the 3D demo works
- Adjust the picture

06

Jupiter display demo suite

Send monochrome phosphor versions of Jupiter's web demos to an MDA display.

The Jupiter demo suite presents a set of small static screens based on the web admin demo: overview, analytics, alerts, orders, invoice, network, settings and warehouse operations. They use the MDA's native phosphor intensity levels; the monitor itself supplies its green color.

06.1

Run the suite

Build and flash the firmware in mode 3, connect the board to Wi-Fi, then send the screens from a computer on the same network:

SH

```
python3 tools/jupiter_display.py 192.168.2.166 --demo all --fps 1 --repeat
```

The script sends compressed 2-bpp frames over the firmware's TCP frame protocol. Stop it with Ctrl-C. To show one screen once, choose a name and omit `--repeat`:

SH

```
python3 tools/jupiter_display.py 192.168.2.166 --demo overview
python3 tools/jupiter_display.py 192.168.2.166 --demo warehouse
```

Available screens: `overview`, `analytics`, `alerts`, `orders`, `invoice`, `network`, `settings`, and `warehouse`.

06.2

Home Assistant and animated screens

A companion sender includes Home Assistant status, energy flow, climate and an animated original retro-futurist executive character. It uses sample data by default; optional Home Assistant integration reads states only and never calls services:

SH

```
python3 tools/ha_display.py 192.168.2.166 --demo all --fps 2 --repeat
python3 tools/ha_display.py 192.168.2.166 --demo house --fps 8 --repeat
```

For live Home Assistant values, create a long-lived access token in your Home Assistant profile and keep it out of shell history and source control. Set it in the environment, then pass the instance URL:

SH

```
export HA_TOKEN='your-long-lived-access-token'
python3 tools/ha_display.py 192.168.2.166 --ha-url http://homeassistant.local:8123
--demo all --repeat
```

The default entity IDs are `sensor.home_temperature`, `sensor.home_humidity`, `sensor.solar_power`, `sensor.grid_power`, `sensor.battery_soc`, `sensor.house_power`, `person.homeowner` and `weather.home`. Override any key with `HA_ENTITY_TEMPERATURE`, `HA_ENTITY_HUMIDITY`, `HA_ENTITY_SOLAR`, `HA_ENTITY_GRID`, `HA_ENTITY_BATTERY`, `HA_ENTITY_HOUSE`, `HA_ENTITY_PRESENCE` or `HA_ENTITY_WEATHER`. The integration calls Home Assistant's read-only `/api/states/<entity_id>` endpoint and polls at a low rate. Without credentials, screens display sample data.

The animated character is original retro-futurist casino-computer artwork: a geometric face, blinking eyes, moving scan line and changing status text. It is not a reproduction of Fallout character artwork or dialogue.

06.3

Preview a screen

Render a screen to a portable graymap without connecting to a board:

SH

```
python3 tools/jupiter_display.py --demo analytics --preview /tmp/analytics.pgm
python3 tools/ha_display.py --demo house --preview /tmp/house.pgm
```

The preview uses gray values for the three display intensities; on hardware those intensities appear as green phosphor. Pillow is optional. If installed, provide the Jupiter SGI Text and SGI Alt font files to use them for body and display text:

SH

```
python3 tools/jupiter_display.py 192.168.2.166 --demo all --repeat \
  --font /path/to/sgi-text.ttf --font-bold /path/to/sgi-alt.ttf
```

Without Pillow or font files, the Jupiter sender uses a compact built-in bitmap fallback. The web demo screens are adapted compositions for 720 × 350, not screenshots of the browser UI; dense controls and color-only distinctions are replaced with readable text, outlines and phosphor intensity.

06.4

Display setup

The board must be running firmware mode 3 and show its IP address. If it is not configured for Wi-Fi, set the SSID and password through `idf.py menuconfig` under the Monochrome menu. See Send frames over Wi-Fi and Frame protocol.

SECTION

Reference

Every command, option and file format.

07

Pins

GPIO to DE-9 pin mapping and the pixel byte format.

SIGNAL	GPIO	DE-9 PIN	NOTES
Video	4	7	Pixel bit. 1 is on, 0 is off.
Intensity	5	6	Bright pixel bit.
H-sync	6	8	18.43 kHz, active high.
V-sync	7	9	About 50 Hz, active low.
Ground	GND	1 and 2	Connect both pins.

07.1

Pixel values

The framebuffer holds one byte per pixel. Bit 0 drives the video pin and bit 1 drives the intensity pin.

BYTE	RESULT
`0x00`	Black
`0x01`	Normal brightness
`0x03`	Bright

Each frame buffer is 720 x 350 bytes (252,000 bytes), indexed as `fb[y * 720 + x]`. Modes 2 and 3 have two frame buffers and modes 0 and 1 have one. The firmware sets the pins in `main/main.c` with `PIN_VIDEO`, `PIN_INTEN`, `PIN_HSYNC` and `PIN_VSYNC`.

SEE ALSO

- Timing
- Wire the monitor

08

Timing

Pixel clock, porches and sync polarity of the 720x350 MDA signal.

The firmware configures the RGB LCD driver (`esp_lcd_rgb_panel_config_t`) in HV mode, with no data enable pin and no pixel clock pin.

SETTING	VALUE
Pixel clock (`pclk_hz`)	16,257,000
Horizontal active (`h_res`)	720
`hsync_front_porch`	18
`hsync_pulse_width`	108
`hsync_back_porch`	36
Horizontal total	882 clocks (18.43 kHz)
Vertical active (`v_res`)	350
`vsync_front_porch`	1
`vsync_pulse_width`	16
`vsync_back_porch`	3
Vertical total	370 lines (about 49.8 Hz)
`hsync_idle_low`	`true` (active-high H-sync)
`vsync_idle_low`	`false` (active-low V-sync)
`data_width`	8, with only the two lowest bits wired

Counting pulses on the sync pins with the pulse counter peripheral gave about 18.4 kHz for H-sync and 50 Hz for V-sync.

SEE ALSO

- How the video works
- Adjust the picture

09

Frame protocol

The TCP and UDP formats that mode 3 accepts, with header layouts, pixel formats and limits.

Mode 3 listens on one port number (default 5050) for both TCP and UDP. Set the port with `MONO_PORT` in `idf.py menuconfig`, under the Monochrome menu. All multi-byte numbers are little endian.

09.1

Pixel formats

FORMAT	VALUE	BYTES PER 720X350 FRAME	PIXEL LAYOUT
8 bits per pixel	0	252,000	One byte per pixel. Bit 0 is video, bit 1 is intensity. The other bits are ignored.
1 bit per pixel	1	31,500	Eight pixels per byte, most significant bit first. 0 is black and 1 is normal.
2 bits per pixel	2	63,000	Four pixels per byte, most significant pair first. 0 is black, 1 is normal, 2 and 3 are bright.

Rows are packed with no padding: 720 bytes, 90 bytes or 180 bytes per row, top row first.

09.2 TCP

The device accepts one client at a time. The client sends frames back to back. Each frame is a header followed by the payload.

OFFSET	SIZE	FIELD	VALUE
0	4	Magic	`MONO`
4	1	Version	1
5	1	Format	0, 1 or 2
6	1	Flags	Bit 0: the payload is zlib-compressed
7	1	Reserved	0

When bit 0 of the flags is set, a 4-byte compressed length follows the header, then that many bytes of zlib data. The data must inflate to exactly the number of bytes in the format table. The compressed length is at most 65,536 bytes. Send larger frames uncompressed.

The device shows each frame when it has fully arrived. It closes the connection after a bad header, an unknown format, a bad length or data that does not inflate to the right size.

09.3 UDP

Each datagram carries some rows of a frame. No connection is needed, and a sender can update only some rows.

OFFSET	SIZE	FIELD	VALUE
0	2	Magic	`MU`
2	1	Version	1

OFFSET	SIZE	FIELD	VALUE
3	1	Format	0, 1 or 2
4	2	Sequence	Frame number. Packets with the same number belong to one frame.
6	2	First row	Row of the first row in this packet, 0 to 349
8	1	Row count	Number of rows in this packet
9	1	Flags	Bit 0: last packet of the frame. Bit 1: the rows are zlib-compressed.

The rows follow the 10-byte header. For a compressed packet the rows are the zlib data, which must inflate to exactly the row count times the row size. A packet carries at most 63,000 bytes of inflated rows. `tools/send.py` keeps each datagram under 1,400 bytes of rows so that it fits in one Wi-Fi packet.

The device ends a frame and shows it when one of these happens:

- a packet with the last-packet flag arrives
- a packet with a different sequence number arrives
- no packet arrives for 50 ms

Rows that a frame does not contain, or that were lost, keep the pixels of the frame that was on screen. A packet with a bad size or an unknown format is dropped.

09.4 One sender at a time

A TCP client and a UDP sender share the frame buffers. While a TCP client is in the middle of a frame, the device drops UDP packets that arrive more than 5 ms later.

09.5 Measured speeds

The device showed these rates on one 2.4 GHz network at a signal strength of -55 dBm, with the demo pattern from `tools/send.py`. The demo is a repeating pattern, so it compresses far better than a photo.

FORMAT	PROTOCOL	COMPRESSION	BYTES PER FRAME	FRAMES PER SECOND
1 bit	UDP	No	31.7 KB	50, no packet loss
2 bits	UDP	No	62 KB	40, about one packet in five lost
2 bits	UDP	Yes	1.8 KB	49.7
2 bits	TCP	No	63 KB	22
2 bits	TCP	Yes	1.5 KB	50
8 bits	TCP	No	252 KB	7.5
8 bits	TCP	Yes	8.4 KB	20 to 26

Random noise does not compress. The sender then sends raw rows, and the device reached about 26 frames per second over TCP and 30 over UDP for 1-bit and 2-bit noise.

SEE ALSO

- Send frames over Wi-Fi
- How the video works

10

Glossary

Terms used in the Monochrome documentation.

Bounce buffer
A pair of small internal RAM buffers that the RGB LCD driver fills from a framebuffer in PSRAM and sends to the pins.

MDA
The IBM Monochrome Display Adapter signal: 720x350 pixels, 18.43 kHz horizontal, about 50 Hz vertical, TTL levels on a DE-9 connector.

Porch
The blank pixels or lines between the end of the picture and the sync pulse (front porch) or after it (back porch).

SECTION

Concepts

How it works and why it is built this way.

11

How the video works

How the ESP32-S3 makes an MDA signal, and the two settings that must not change.

The ESP32-S3 has an RGB LCD peripheral that reads pixels by DMA and outputs them with sync pulses. The firmware uses it as an MDA video source. The monitor takes video, intensity, H-sync and V-sync, and needs no pixel clock, so the firmware sets `pclk_gpio_num` and `de_gpio_num` to -1.

11.1

Frame buffers

The 720 x 350 frame buffer is 252,000 bytes. The firmware offers two layouts:

LAYOUT	USED BY	FRAME BUFFER	BOUNCE BUFFER
Internal RAM	Modes 0 and 1	One, in internal RAM	None
PSRAM	Mode 2	Two, in PSRAM	Two of 35 lines each, in internal RAM

With the PSRAM layout the driver copies the frame buffer into the *Bounce buffer* pair in small chunks, and the DMA reads the bounce buffers. The PSRAM layout leaves most of the internal RAM free and lets a program draw into one buffer while the DMA shows the other.

11.2

Keep the bounce chunk count even

The number of bounce chunks per frame must be even. The firmware checks this at compile time in `main/main.c`.

BOUNCE SIZE	CHUNKS PER FRAME	RESULT ON THE TEST MONITOR
10 lines	35 (odd)	Picture bounced vertically and showed doubled lines
70 lines	5 (odd)	Picture steady, but bands of 70 lines repeated or swapped
35 lines	10 (even)	Steady and correct

The V-sync interval measured 20,072 to 20,073 microseconds in every frame in all of these cases, so the sync timing was never the problem. The scrambled bands show that the data order was wrong. The driver alternates two bounce buffers and restarts the DMA from the first buffer at each V-sync. An odd chunk count leaves the two out of step at the end of a frame. This explanation fits the three results above. The sizes 25 lines and 5 lines (also an even count) were not tested.

The options `CONFIG_LCD_RGB_RESTART_IN_VSYNC`, `CONFIG_LCD_RGB_ISR_IRAM_SAFE` and 64-byte data cache lines did not fix the odd-count cases, and the firmware does not use them.

11.3 Reuse a frame buffer only after the driver latches the new one

With two frame buffers, a program draws into the hidden one and calls `esp_lcd_panel_draw_bitmap` with it. That call does not wait. The driver switches to the new buffer once per frame, when it has copied the last chunk of the frame into the bounce buffers. It then calls `on_frame_buf_complete`. The old buffer is free to overwrite only after that call.

Waiting one V-sync after the flip is not enough. If the flip happens in the last few dozen lines of a frame, the driver reads the old buffer for one more frame, and a program that writes into it shows torn lines. The firmware counts the `on_frame_buf_complete` calls in `g_latched` and waits for the count to advance before it writes (`wait_latched` in `main/main.c`).

The Wi-Fi receiver exposed this. Its first version blocked for up to a frame after each flip, so most UDP packets overflowed the receive queue (about 9 of 50 packets per frame arrived). Waiting lazily, only when the next frame starts, raised that to 40 to 49 of 50 and removed the visible tearing.

11.4 Keep V-sync active low

MDA uses an active-low V-sync and an active-high H-sync. When the firmware sent an active-high V-sync, the monitor used for development changed its picture size and brightened the raster. That suggests the monitor selects a CGA-style mode from the V-sync polarity. The firmware sets `vsync_idle_low = false` to keep the MDA mode.

11.5 Hercules

A Hercules card sends the same kind of signal: the same DE-9 pins, the same TTL levels and the same sync polarities. A Hercules picture is 720x348 at a pixel clock of about 16 MHz. A monitor that accepts MDA accepts it too. The firmware does not emulate the Hercules memory layout.

SEE ALSO

- Frame protocol
- How the 3D demo works
- Timing

■ *MDA*

12

How the 3D demo works

A software renderer that draws a shaded torus knot at 49 frames per second on both cores.

Mode 2 runs a software 3D renderer in `main/knot.c`. It draws a lit torus knot at 49 frames per second, which is the display's own rate of about 49.8 Hz.

12.1

The scene

ITEM	VALUE
Mesh	A (2,3) torus knot, 64 segments along the curve and 12 around the tube: 768 vertices, 1,536 triangles
Triangles drawn per frame	About 810 after back-face culling
Shading	Per-vertex diffuse and specular light, interpolated across each triangle (Gouraud)
Light	Orbits the knot
Depth cue	Far parts are dimmer
Background	140 drifting single-pixel stars

12.2

Pixel output

The monitor shows three levels: black (`0x00`), normal (`0x01`) and bright (`0x03`). The renderer works in 8-bit brightness and uses an 8x8 ordered (Bayer) dither to choose between the two nearest levels for each pixel. It scales x by 1.54 so that shapes stay round, because pixels on a 4:3 tube showing 720x350 are taller than wide.

12.3

Each frame

1. **Prepare (core 0, about 2.2 ms).** Rotate the vertices and normals with one combined matrix, project them, light them, drop back-facing triangles and sort the rest far to near with a bucket sort.
2. **Rasterise (both cores, about 15 ms each).** The screen is split into ten strips of 35 lines. Each core takes the next unclaimed strip, draws it into a small buffer in internal RAM, and copies it into the PSRAM frame buffer. A core that is busy with display interrupts takes fewer strips.
3. **Flip.** The firmware calls `esp_lcd_panel_draw_bitmap` with the finished buffer, which tells the driver to scan it out from the next frame. It then waits until the driver has switched to the new buffer before it reuses the old one. See How the video works.

Frames take 17 to 18 ms to render, against 20.07 ms between V-syncs.

12.4

What made it fast

The first version ran at 25 fps with 27 to 29 ms per frame. These changes brought it to 49 fps:

CHANGE	EFFECT
Draw into internal-RAM strips and copy to PSRAM	Fewer slow PSRAM writes
Halve the mesh (128x14 to 64x12 segments)	Fewer, larger triangles and about half the per-row setup cost; prepare dropped from 5.3 ms to 2.2 ms
Reciprocal table instead of an integer divide per scanline	Cheaper span setup
Strips shared between cores through an atomic counter	Both cores stay busy
Brightness limited to 2..253	The pixel loop needs no clamps

SEE ALSO

- How the video works
- Choose a demo

13

Monochrome visual language

Typography and line-art rules for bringing Jupiter's designer fonts and shapes to a green-phosphor MDA display.

Monochrome inherits Jupiter's typography and graphic vocabulary. The MDA monitor supplies the color: green phosphor on a dark screen. There is no application palette to design, tune, or reproduce. The firmware's pixel values express **intensity**, not different colors.

13.1

Inherit from Jupiter

Use the type and shape language already defined by the Jupiter design system:

JUPITER ROLE	DISPLAY USE	RENDERING RULE
SGI Alt	Screen titles and short, prominent words	Use a purpose-made pixel glyph set or atlas; preserve the typeface's distinctive forms rather than stretching a web font at runtime.
SGI Text	Instructions and readable supporting text	Keep text short, left aligned, and legible at the chosen raster size.
JetBrains Mono	IP addresses, ports, counters, timings and protocol values	Use fixed-width glyphs so columns and changing values stay aligned.

JUPITER ROLE	DISPLAY USE	RENDERING RULE
Jupiter line graphics	Rules, icons and illustrations	Start from the shared vector artwork; rasterize to the target grid and preserve its silhouette and proportions.

The SGI fonts are designer typefaces, not a requirement to draw arbitrary TrueType outlines in the firmware. Font rasterization and atlas generation belong in the Jupiter build pipeline. Firmware should consume the generated, display-ready glyph data; it should not carry a second hand-designed font or copies of the font source.

Likewise, source line art from Jupiter's graphics package rather than inventing a parallel icon style. A shape intended for the MDA should have a monochrome raster output generated from that shared artwork, not a locally redrawn substitute.

13.2 The display's visual grammar

The signal is 720 × 350 pixels at about 50 Hz. The framebuffer's existing values are:

PIXEL VALUE	MEANING	USE
`0x00`	Phosphor off	Background and clear space
`0x01`	Normal phosphor intensity	Body text, fine outlines, dividers and secondary marks
`0x03`	Bright phosphor intensity	Heading emphasis, key values and the most important parts of a shape

Use brightness sparingly to establish hierarchy. Do not assign semantic color meanings to intensity: success, warning and error must be communicated with words or distinct shapes, not a color mapping. The display has no color-management or light/dark theme switch.

Jupiter's visual character is quiet, precise and line-led. On this display, favor open space, thin outlines, simple geometric frames and a small number of strong typographic anchors. Prefer an outline or continuous stroke over a solid block. Use filled areas only where the image needs a clear signal or a deliberate large-scale contrast.

13.3 Raster and layout rules

- Design against the real 720 × 350 framebuffer, not a stretched 4:3 mockup. The CRT's pixels are not square; firmware geometry accounts for this with a horizontal aspect correction of about 1.54.
- Keep important content inside a deliberate inset. The calibration pattern's 72-pixel columns and 35-line rows are test geometry, not a mandatory application grid.
- Snap horizontal and vertical strokes to whole pixels. Avoid subpixel line widths and fine detail that disappears when reduced to the target raster.
- Preserve the source proportions of a Jupiter glyph or graphic. Scale deliberately, then inspect the resulting pixels; never squash artwork to fit a box.
- Keep neighboring strokes distinct. Dense details, tight counters and closely spaced parallel lines can merge on a phosphor display.

- Use fixed-width type for changing numeric values and reserve the brightest intensity for a small number of focal elements.
- Treat animation as information. Keep its silhouette readable at roughly 50 frames per second and avoid motion that adds no meaning.

13.4 Shared source and generated outputs

The source of truth stays in `belikh/design`: SGI Alt, SGI Text and JetBrains Mono for typography; Jupiter graphics for line art. Display-specific glyph bitmaps and shape rasters are **generated derivatives**, not new sources of design. Keep generated build artifacts out of this repository unless the Jupiter publishing workflow explicitly requires a checked-in fixture.

The current firmware has a small 5×7 digit font for its network status screen and can receive 1-bit or 2-bit frames over the network. Those are implementation capabilities, not the finished Jupiter type system. The next integration step is to add a Jupiter-generated bitmap-font and monochrome-shape target, then consume its output here. Until that target exists, do not claim that the existing firmware renders the SGI fonts or Jupiter graphics directly.

13.5 Review checklist

1. Is the artwork recognizably from Jupiter's type and graphic vocabulary?
2. Does it remain clear using only phosphor off, normal intensity and bright intensity?
3. Are meaning and status understandable without color?
4. Do letters retain their intended shape and readable spacing at the actual raster size?
5. Do thin strokes survive on the real monitor without crowding or filling in?
6. Is the aspect-corrected result checked on hardware, not only in a square-pixel preview?