Backend — Commands

Location: src-tauri/src/commands/

Tauri commands are Rust functions exposed to the frontend via *invoke*. All commands are registered in and live in the module. Each page below covers one file.

rover_state.rs — Rover Mode State

> The state described here MIGHT be subject to change

Manages the three global boolean flags that track the rover's current operating mode. The state is held in a `RoverState` struct registered as Tauri managed state (initialised in `lib.rs`), so it persists for the lifetime of the application.

State fields

State	Default	Description
`drive_manual_mode`	`true`	Whether the rover driving is in manual control
`arm_manual_mode`	`true`	Whether the arm is in manual control
`pickup_mode`	`false`	Whether the rover is in driving or pickup mode

Commands

`get_state(state_type: StateType) → bool` Returns the current value of the requested state flag. Called on page mount to sync the UI with the actual rover state.

`set_state(state_type: StateType, value: bool)` Sets a state flag. Called by the frontend when the operator switches modes (e.g. toggling pick-up mode on the dashboard).

`StateType` is an enum with variants: `DriveManual`, `ArmManual`, `Pickup`.

file_management.rs — Persistent File Storage

Handles all file I/O for the application. Files are stored inside Tauri's `app_data_dir`, which is platform-specific (e.g. `%APPDATA%\base_station` on Windows). Three subdirectories are used:

Directory	Purpose
`tasks/`	Saved task plan files
`images/`	Snapshots captured from video feeds
`maps/`	Imported map files

These directories are created automatically on startup by `ensure_storage_dirs_internal()`.

Commands

`save_task_file(file_name, data, directory)` Writes raw bytes to a file in the given subdirectory. Used to persist task plans.

`list_task_files(directory) → Vec<String>` Returns a list of filenames in the given subdirectory. Returns an empty list if the directory doesn't exist yet.

`read_task_file(file_name) → Vec<u8>` Reads a file from the `tasks/` directory and returns its raw bytes.

`delete_one_file(directory, file_name)` Deletes a single named file from the given subdirectory. Does nothing if the file doesn't exist.

`delete_all_task_files(directory)` Removes all files in a subdirectory by deleting and recreating it.

`import_map_file(directory)` Copies a file from an arbitrary path on the filesystem into the `maps/` subdirectory. Used when the operator imports a new map via the file picker. The original filename is preserved.

`save_snapshot(port, file_name)` Captures a single JPEG frame from an MJPEG stream (given by its localhost URL/port) and saves it to the `images/` directory as `{file_name}.jpg`. Used in the Science task to photograph samples. It scans the raw HTTP stream for the JPEG start marker (`0xFF 0xD8`) and end marker (`0xFF 0xD9`), extracting the first complete frame. Has a 5 MB safety limit per frame.

gstreamer.rs — Video Streaming

Receives H.264 video from the rover over UDP, decodes it, and serves it as MJPEG over HTTP so the frontend can display it in `<img>` tags.

Pipeline per camera

```
udpsrc (UDP port) → rtpjitterbuffer → rtph264depay → avdec_h264 → videoconvert → jpegenc → appsink
```

Each decoded JPEG frame is placed into a shared `FrameBuffer` (`Arc<Mutex<Option<Bytes>>>`). A separate async HTTP server (using `warp`) reads from that buffer and streams it as `multipart/x-mixed-replace` — the standard MJPEG format.

Port mapping

UDP input port	HTTP output port	Camera
4500	5000	Depth / front camera
4501	5001	Secondary camera
4502	5002	Arm camera

Feed health monitoring

A background task (`watch_feed_health`) polls each stream every 500ms. If no frame has been received within 2 seconds, the stream is considered stale. The backend emits a `camera-feed-status` Tauri event to the frontend with the payload:

```
{ "port": 5000, "stale": true }
```

The frontend listens for this event to show feed status indicators.

> **For development without rover hardware**, run the `fake_camera_gstreamer`. For instructions see Common Operations.

network.rs — UDP & Dummy Simulator

> This file is work in progress

Exposes commands related to the UDP connection and the development simulator.

Commands

`send_ping_cmd(packet_type)` Intended to send a ping packet to the rover over UDP. Currently logs to console — full implementation pending.

`start_dummy_streams()` Starts the full multi-stream simulator. Sends fake protobuf packets to `127.0.0.1:9000` mimicking live rover data. Useful for UI development without hardware. Simulator config:

- Jitter: 30ms
- Simulated packet loss: 2%

`start_detection_sim()` Starts a single stream for the detection object stream dummy data.

`stop_dummy_streams()` Signals the running simulator to stop by setting a shared cancellation flag.

The `DummyStreamHandle` struct (managed Tauri state) holds the cancellation handle so any command can stop the simulator.

controller.rs — Gamepad Input

Runs a background listener for gamepad input using the `gilrs` library, translating button and axis events into UDP packets sent to the rover. All dispatching is gated on the current mode (drive vs. pickup) and whether the relevant manual mode is active.

Modes

The controller operates in one of two top-level modes, toggled with the **Start** button:

Mode	Active when	Controls
Drive	`pickup_mode = false`	Left stick (forward/turn), triggers (brake)
Pickup	`pickup_mode = true`	Both sticks (X/Y/rotate/flick), D-pad (Z), triggers (gripper)

Within each mode, the **Select** button toggles the relevant manual mode (`drive_manual_mode` or `arm_manual_mode`). Commands are silently suppressed when the associated manual mode is inactive.

Threads

Three concurrent threads are spawned on startup:

`start_controller_listener()` — Entry point. Spawns all threads and owns the shared `CommandState`.

Event thread — Polls `gilrs` at ~125 Hz (8 ms sleep) and routes each `Event` to the appropriate handler. Holds the `shared` mutex only for the duration of each state update.

Heartbeat thread — Wakes every 2 seconds and re-sends the current state (drive axes + brake, or arm + brake). Ensures the rover never silently drifts from its commanded state if packets are dropped.

Ramp threads — Spawned on demand when a ramped button (D-pad up/down, left/right trigger in pickup mode) is pressed. Ticks at ~60 Hz and increments the axis value from `0.0` toward `±1.0` over `RAMP_DURATION_SECS` (1.0 s). The thread exits when the direction is set back to `0.0` on button release.

Button mapping

Button	Drive mode	Pickup mode
Start	Toggle pickup mode	Toggle pickup mode

Select	Toggle `drive_manual_mode`	Toggle `arm_manual_mode`
Left trigger (LB)	Toggle latching brake	Ramp gripper speed → close (−1.0)
Right trigger (RB)	—	Ramp gripper speed → open (+1.0)
Left trigger 2 (LT)	Toggle latching brake	—
Right trigger 2 (RT)	Momentary brake (hold)	—
D-pad up	—	Ramp Z → up (+1.0)
D-pad down	—	Ramp Z → down (−1.0)

Axis mapping

Axis	Drive mode	Pickup mode
Left stick Y	Forward / backward	Flick
Left stick X	Turn	Rotate
Right stick X	—	End effector X (left/right)
Right stick Y	—	End effector Y (forward/backward)

All analog axes pass through a deadzone (`±0.05`) and are only dispatched when the change from the last-sent value exceeds `AXIS_CHANGE_THRESHOLD` (0.05). Z and gripper speed are not axis-driven — they are ramped from button presses.

Packets sent

Packet	When
`BasestationManualDrive`	Drive mode, on axis change or heartbeat
`BasestationManualBrake`	Drive mode, on brake toggle/press/release or heartbeat; also sent continuously (engaged) during pickup heartbeat
`BasestationManualArmMovement`	Pickup mode, on any arm state change or heartbeat

All values are scaled from `[−1.0, 1.0]` to the full `sint32` range before transmission.

Constants

Constant	Value	Purpose
`AXIS_CHANGE_THRESHOLD`	`0.05`	Deadzone boundary and minimum delta before a packet is sent
`HEARTBEAT_INTERVAL`	`2 s`	How often state is re-sent without an input event

`RAMP_DURATION_SECS`	`1.0 s`	Time for a ramped axis to travel from `0.0` to `±1.0`
`RAMP_TICK_MS`	`16 ms`	Ramp thread tick interval (~60 Hz)
`MOMENTARY_BRAKE_DURATION`	`500 ms`	Defined but unused — auto-release was commented out

checks.rs — Diagnostics

Two utility commands for diagnostics and maintenance.

`ping()` Prints `"PING FROM RUST"` to the console. Used to verify the Tauri bridge is working.

`clear_cache()` Deletes all contents of the `base_station/` folder inside the system cache directory. Exposed as a callable command so the frontend can trigger a manual cache wipe.

`clear_cache_on_startup()` is the internal (non-command) version called automatically every time the app launches.

> **Note for future developers:** Clearing the cache on startup is necessary, without doing so the video feed and other assets will not load in some devices.

load_model.rs — 3D Model Loading

Handles loading of 3D model files bundled with the application.

`load_model(path) → Vec<u8>` Reads a model file by filename and returns its raw bytes to the frontend. In debug builds, models are loaded from `src-tauri/models/`. In release builds, they are loaded from Tauri's resource directory (where they are bundled via `tauri.conf.json`).

`debug_resource_dir()` Returns the resource directory path and its contents as a string. Used during development to verify that model files are bundled correctly.

> **Note for future developers:** Webview technically do this automatically from the frontend but it errors out so the custom function is necessary.

rover_commands.rs — Rover Science Commands

> This file is work in progress

These commands are called by the frontend to request measurements or send data to the rover. All four are currently partially stubbed — they simulate a 1-second rover response delay and return dummy values, but the actual UDP dispatch logic is ready to be wired in.

`request_coordinates() → (i16, i16)` Requests the rover's current GPS coordinates. Returns a `(latitude, longitude)` tuple.

`request_weight() → i16` Requests the weight of a collected rock sample from the rover's load cell. Returns a weight value in grams.

`request_measurement(camera1, x1, y1, camera2, x2, y2) → i16` Requests a stereo distance measurement. The frontend passes two pixel coordinates (one per camera) and the rover computes the real-world distance to that point. Returns the distance in a unit TBD.

`send_pixel(camera, x, y)` Sends a single pixel coordinate from the frontend to the rover (e.g. operator clicking on an object in the video feed). Used to direct the rover's attention or arm toward a specific point in the image.

map_commands.rs and map_processor.rs — 3D Map Rendering & Coordinate Transforms

`map_processor.rs` + `map_commands.rs` — 3D Map Rendering & Coordinate Transforms

Converts 3D map files into top-down orthographic PNG images coloured by height (Z), and exposes coordinate transforms so the frontend can convert pixel clicks back to real-world metres. The pipeline runs on a blocking thread to avoid stalling the async runtime.

Supported formats

Extension	Library	Notes
`.obj`	`tobj`	OBJ Y-up convention remapped: world X = OBJ X, world Y = OBJ Z, height = OBJ Y
`.las`	`las`	X/Y/Z read directly from point records
`.laz`	`las`	Same as LAS (compressed)

Commands

`render_map(filename)` → `MapMeta` Renders a 3D map file to a top-down PNG. The source file must already exist in `<appDataDir>/maps/`. The output is written to the same directory as `<stem>_preview.png`. The image is sized to a 2048 px longest edge, aspect ratio preserved. Heavy work is offloaded via `spawn_blocking` so the async runtime is never blocked. Returns a `MapMeta` struct the frontend uses for pixel→world transforms.

`pixel_to_world(px, py, meta)` → `(f64, f64)` Converts a 2D pixel coordinate (origin bottom-left, X right, Y up) to real-world metres. Expects coordinates in the displayed image's frame — rotation, if any, must be accounted for by the frontend before calling this. The formula is simply `world = pixel × metres_per_pixel`, with the world origin anchored to the bottom-left corner of the image.

`MapMeta` fields

`MapMeta` is serialised and returned to the frontend after every `render_map` call.

Field	Type	Description
`img_width`	`u32`	PNG width in pixels (post-rotation)
`img_height`	`u32`	PNG height in pixels (post-rotation)
`world_x_min`	`f64`	Real-world X at the left edge (currently always `0.0`)
`world_y_min`	`f64`	Real-world Y at the bottom edge (currently always `0.0`)
`metres_per_pixel`	`f64`	Scale factor for pixel→world conversion
`format`	`String`	Source format detected (`"obj"`, `"las"`, `"laz"`)
`rotated`	`bool`	`true` if the image was rotated 90° to landscape

Rendering pipeline

`process_map()` runs the full pipeline in five stages:

1. **Load** — Parse the source file into a flat list of `Point3D` structs (`x`, `y`, `z` in world space).
2. **Bounding box** — Compute `x_min/max`, `y_min/max`, `z_min/max` over all points. World width and height must be non-zero or an error is returned.
3. **Rasterise** — Map each point to a pixel coordinate. Where multiple points land on the same pixel, keep the highest Z (i.e. the sky-facing surface wins). Pixel dimensions are derived from `img_size` (2048) with the aspect ratio preserved.
4. **Gap fill** — Run a two-pass nearest-neighbour distance transform (`fill_gaps`) to fill pixels that received no points. The forward pass sweeps top-left → bottom-right (checking left and top neighbours); the backward pass sweeps bottom-right → top-left (checking right and bottom neighbours). Each unfilled pixel inherits the Z of its closest filled neighbour, eliminating stripe artifacts.
5. **Colour + save** — Each pixel's Z is normalised to `[0.0, 1.0]` and passed through a five-stop colour ramp (`height_color`): deep blue → cyan → green → yellow → red. If all points share the same Z (flat terrain), mid-green is used. The image is rotated 90° if height exceeds width (to keep the longest edge horizontal), then saved as PNG.

Height colour ramp

Normalised Z	Colour
0.0	Deep blue
0.25	Cyan
0.5	Green
0.75	Yellow
1.0	Red

Error conditions

Condition	Error returned
Unsupported file extension	`"Unsupported format: .{ext}"`
File parsed but empty	`"File parsed but contained no points."`
All points collinear in X or Y	`"All points are collinear — cannot build a 2D map."`
Source file missing at invoke time	`"File not found: {path}"`
PNG write failure	`"Failed to save PNG: {e}"`