AURORA

Version 14bd812

Table of Contents

AURORA

AURORA (**AU**xspace **R**eal-time **O**nboard **R**ocket **A**vionics) is a Zephyr RTOS-based avionics firmware for rockets, developed by Auxspace e.V..

It manages sensor data, flight state detection, and pyrotechnic ignition across multiple interconnected boards communicating over CAN bus.

AURORA itself is made up of several key components:

- applications
- boards
- drivers
- libraries
- tests
- tools

As depicted in the Application Development guide from the Zephyr Project Documentation, AURORA acts as a Zephyr freestanding application.

AURORA Libraries and Drivers

The core parts of AURORA are its libraries and drivers, that can be used by any Zephyr Application with simple `#include <aurora/...>` statements, `devicetree overlays` and `Kconfigs`.

Libraries are located at `lib` while drivers can be found in the `drivers` directory.

Their public APIs are documented in the API-Reference, including the library and the driver APIs.

AURORA Applications

Applications use libraries and drivers to function. An example is the `sensor_board` application, that acts as a flight computer using IMU and barometric pressure data and deploying a parachute when rocket apogee is detected.

Since application's use cases vary, `Kconfigs` and `Devicetree Overlays` are used to customize the application's function for each board. Some boards may have additional pyro ignitions or the capability to signal state machine changes on a buzzer while others perform communication tasks over CANbus and use redundant sensor data (e.g. High G Accelerometer combined with the standard 6 or 9-DoF IMU).

Project

Getting Started

Prerequisites

In the AURORA git repository's README and the Getting Started Guide guide from `zephyr` is everything you need to install first.

> **Note**
>
> Tagged builds are published on the GitHub releases page.

AURORA is built inside a `west` workspace. The directory layout is:

```
zephyr_workspace/
├── aurora/          ← this repository
├── modules/
├── .west/
└── zephyr/
```

All `west` commands must be run from the `aurora/` directory.

Building

Build the primary `sensor_board` application for one of the supported boards:

```
# RP2040 (primary target)
west build -b sensor_board_v2/rp2040 sensor_board

# RP2350 RISC-V
west build -b sensor_board_v2/rp2350a/hazard3 sensor_board

# ESP32-S3 Micrometer board
west build -b micrometer/esp32s3/procpu --sysbuild sensor_board/
```

> **Note**
>
> Build output is located at `build/zephyr/zephyr.uf2` and `build/zephyr/zephyr.elf` for the default builds, while `--sysbuild` generates subdirectories for `MCUBoot` and the application at `build/mcuboot/zephyr/zephyr.elf` and `build/sensor_board/zephyr/zephyr.signed.bin`.

Interactive Kconfig

```
# Using west
west build -b \<BOARD\> -t menuconfig \<APPLICATION\>

# Using the wrapper
./run.sh -b sensor_board_v2/rp2040 menuconfig
```

Docker Container

```
# Open a shell inside the dev container
./run.sh -b sensor_board_v2/rp2040 shell
```

Flashing

Zephyr's West

```
west flash
```

> **Note**
>
> Boards like the esp32s3 that use an interactive download mechanism need to trigger the download mechanism **twice** for sysbuild binaries, since flashing multiple binaries require multiple reboots.
>
> In case of the esp32s3, start by booting in download mode, run the flashing command and wait for the first binary to be flashed.

Reboot the board by cutting and re-establishing power to it, while still asserting the switch that activates download mode.

Repeat until the flashing command has finished.

OpenOCD (CMSIS-DAP)

```
# e.g. sensor_board_v2/rp2040
sudo openocd -f interface/cmsis-dap.cfg -f target/rp2040.cfg \
  -c "adapter speed 5000" \
  -c "program build/zephyr/zephyr.elf verify reset exit"
```

USB (on sensor_board_v2 via RPi Pico BOOTSEL mode)

Hold BOOTSEL while connecting the board, then copy the UF2 file:

```
cp build/zephyr/zephyr.uf2 /media/${USER}/RPI-RP2
```

Configuration

All AURORA features are toggled and tuned through Kconfig. Use `west build -b <board> -t menuconfig` or `./run.sh -b <board> menuconfig` to browse options interactively.

By hitting `D`, menuconfig supports minimal config saving. After saving a minimal config, run `tools/sync_defconfig.py` to merge the minimal config automatically into the desired project.

⚠ Warning

sync_defconfig.py is not an official part of zephyr, so use the script at own risk! (The high risk being that a config might be different from what you were expecting)

Testing

AURORA uses the Zephyr ztest framework and the twister test runner. Unit tests run on `qemu_x86` and require the `x86_64-zephyr-elf` toolchain.

Running Tests

```
# Run all unit tests
west twister -T tests -v --inline-logs

# Run a single test suite by path
west twister -T tests/lib/state -v --inline-logs

# Build-verify sensor_board against all supported boards
west twister -T sensor_board -v --inline-logs --integration
```

Test Layout

Each test suite lives under `tests/` in a directory that mirrors the library it covers. A `testcase.yaml` file alongside the test source tells Twister how to build and run the suite:

```
tests/
└── lib/
    └── state/
        ├── testcase.yaml
        └── src/
            └── main.c
```

Writing Tests

Tests use the `ZTEST_SUITE` / `ZTEST` macros. Per-test setup and teardown callbacks can be registered to initialize and reset state:

```
ZTEST_SUITE(my_suite, NULL, NULL, before_fn, after_fn, NULL);

ZTEST(my_suite, test_something)
{
    zassert_equal(result, expected, "message");
}
```

See `tests/lib/state/src/main.c` for a complete example that exercises the simple state machine through its full flight sequence.

Applications

As stated on the main page, AURORA contains out of the box applications for model rocketry.

The application code lives in the top-level directories of the `aurora` repository, e.g. `sensor_board`.

Sensor Board

Source: `sensor_board`

The sensor board application is a simple flight computer for model rocketry.

AURORA is built to run on multiple interconnected PCBs that communicate over CAN bus, i2c or other fieldbus technologies. `sensor_board` manages the IMU and barometric sensor data, uses the flight state machine, pyrotechnic ignition, hardware notification and data logging and is designed to deploy a parachute when the model rocket reaches apogee.

Operating the Board in the Field

This section walks through what actually happens on the launch site, from the moment you flip the power switch to the moment you walk back from recovery. It is written for the person standing next to the rocket, not for the firmware developer. For the internals, see the sections further down.

What the board is telling you

The board talks to you through two channels: a **buzzer** and one or more **status LEDs**. There might be no screens on the launchrail, so it is worth getting used to both before the first real flight. The two backends signal the same flight states, so you can rely on whichever is more practical for the conditions (LED in a noisy crowd, buzzer once the rocket is out of sight).

Buzzer, in broad strokes:

- **Boot sound**: short jingle the moment the board is powered on. If you don't hear this, the board is not running.
- **Calibration started beeps**: two short, low beeps once the board begins its stationary IMU calibration window (it runs in the background during `IDLE`, so this normally follows shortly

after the boot sound, as soon as the rocket is standing still). Same pitch as the completion tone below, so both read as "calibration"; the rhythm tells them apart (two short = started, one long = done). You only hear this **once per `IDLE` cycle**: if the board restarts its window because you bumped the rail, it does *not* beep again. Hearing it a second time means the board went back to `IDLE` (e.g. it disarmed).

- **Calibration complete tone**: a single confirmation tone once IMU bias calibration finishes (it runs in the background during `IDLE`). Pyros are **not** yet live until the `ARMED` beep below.
- **Calibration-finished tone, then arming beep**: once calibration converges the board immediately plays a confirmation tone followed by a higher, longer beep as it moves into `ARMED` — that second beep is your "pyros are now live" cue.
- **Landed song**: a longer tune that loops after touchdown and keeps going until you disarm or power down. This is your recovery beacon and shall help you find the rocket in a corn-field or a difficult place.

Status LED, in broad strokes:

- **Boot flash**: solid ON for half a second, then off. Pairs with the boot jingle and confirms the board has come up.
- **IDLE**: short, sparse pulses (about twice a second). The rocket is disarmed and safe to handle.
- **(No separate CALIBRATING state)**: IMU bias calibration runs in the background while the board remains in `IDLE`; pyros are **not** yet live. The LED keeps the `IDLE` pattern throughout, so the calibration cues are on the buzzer only.
- **ARMED**: even, steady blink (long ON time, unlike the sparse `IDLE` pulse). Calibration is done and pyros are live. If you see this from the launchpad walk-back, the board is ready.
- **In-flight (BOOST through REDUNDANT)**: LED stays dark. This is intentional. It saves battery and avoids optical noise during flight.
- **LANDED**: long pulses (mostly ON, brief OFF). Pairs with the landed song as a visual recovery beacon.
- **ERROR**: solid ON without blinking. Service required.

Tip

The full pattern tables are in Buzzer Patterns and LED Patterns.

Note

The LED backend depends on the data logger being enabled. When the data logger is disabled, the LED stays dark. The buzzer is unaffected.

Step-by-step commissioning

The following sequence assumes a board that has already been built and flashed, with a charged battery and a µSD-card installed. Hardware-specific arming mechanisms (key switch, button, plug, ...) differ per board. Check the board's own page under Supported Boards for the physical detail.

1. **Pre-flight check on the bench.** Battery charged, µSD-card inserted, pyro channels wired but *not yet connected to igniters*. Confirm the rocket is in the disarmed state before going anywhere near the pad.

2. **Place the rocket on the launchrail.** Do this *before* arming. The board uses its first stable orientation as the reference for "vertical", so any leaning or repositioning after arming will throw off the calibration. When you move the rocket in an armed state, make sure to disarm and rearm again when the rocket is in its final position before liftoff.

3. **Power on.** You should hear the boot sound and see the LED flash solid for about half a second, then drop into the slow IDLE blink. The board is now running but is still in the IDLE state. It will not react to motion yet. Once the rocket is standing still you will hear the two short calibration-started beeps: the board has begun its stationary calibration window. It runs in the background during `IDLE`, so this happens whether or not you have armed yet.

4. **Connect the igniters.** Only do this with the board powered on but still disarmed, following your range's safety procedure.

5. **Arm the board.** Trigger the arming mechanism (the exact action is board-specific). The board stays in `IDLE` until calibration has finished, so pyros are not yet live at this point. If you already heard the calibration-finished tone in step 3, arming moves it to `ARMED` right away.

6. **Hold still.** Don't bump the rail, don't walk into the rocket, don't re-aim. If the board detects motion during calibration (still `IDLE`) it discards the in-progress window and restarts automatically from the next still sample — no need to disarm and re-arm for a single bump, though sustained shaking (a gusty rail) will keep resetting the window and delay completion. Calibration also finishes early on its own once the estimate stops improving, rather than always waiting for a fixed duration.

7. **Wait for the ready tone.** Once you hear the calibration-finished tone followed by the higher arming beep, and see the LED switch to the slower, even `ARMED` blink, the board has finished calibrating, pyros are now live, and it's waiting for liftoff. You can now clear the pad.

8. **Need to abort or re-aim?** Disarm the board. It will return to IDLE. You can re-arm as many times as you like. The state machine just cycles back into `IDLE` each time, waiting on calibration if needed, which is by design. There is no "wasted" arming attempt.

9. **Launch.** The state machine takes over from here: BOOST, BURNOUT, APOGEE (drogue parachute deployment), descent, MAIN, REDUNDANT and finally LANDED. See the state machine diagram for the full transition graph.

10. **Recovery.** When the board detects landing, it starts playing the landed song and the LED switches to the long-pulse landed pattern. Walk in the direction of the sound; if it is a bright day, the LED can also help once you are within line of sight. Once you've found the rocket, disarm or power down the board to silence it.

> **Note**
>
> Usually it is not a problem, but removing the µSD card from the board should only be done when the board is disconnected from the battery.

Things that commonly go wrong

- **Calibration takes a long time / never finishes.** The board automatically restarts the calibration window whenever it detects motion, so persistent wind or vibration on a tall rail can keep it resetting indefinitely. It will always finish eventually (there's a sample-count safety ceiling), but a long stall is a sign the rail isn't stable enough yet. Disarm, wait for the rail to settle, and re-arm if it's taking too long for your launch window.

- **Board disarms by itself.** The rail is tilted past the disarm angle threshold (see Configuration below). Either re-aim the rail or relax the threshold for the campaign.

- **No landed song after recovery.** Either the battery is empty, the µSD card filled up mid-flight and the board faulted, or the landing was not detected (e.g. it is still drifting under canopy in a tree). Check power and the data log before assuming a software issue.

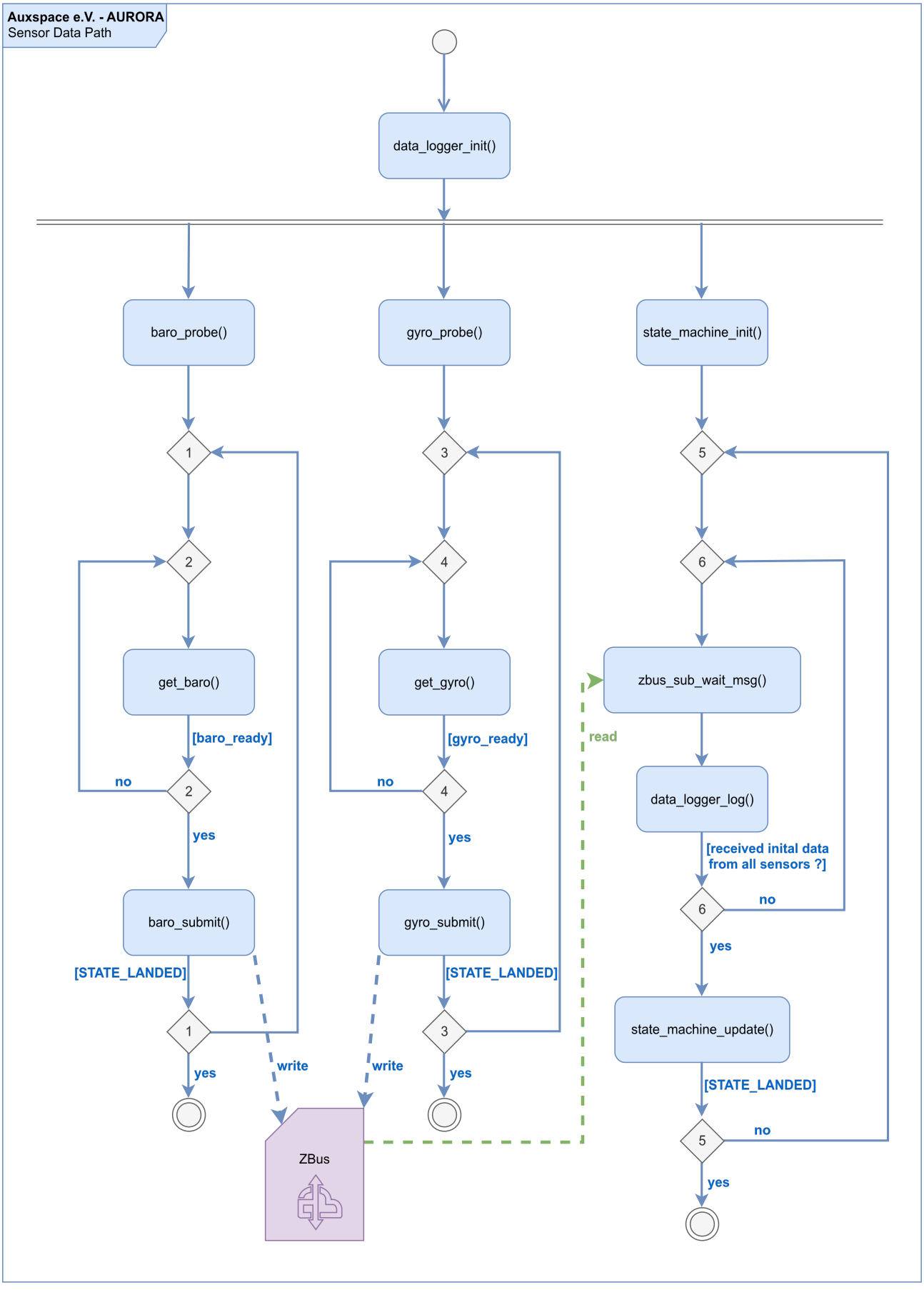
- **LED stays solid ON without blinking.** That's the ERROR pattern, not a ready indicator. The board has hit an unrecoverable fault. Disarm, power down, pull the µSD card, and check the log before flying.

- **LED never lights up at all.** The LED backend depends on the data logger; if the logger is disabled (or failed to mount the µSD card), the LED stays dark even though the rest of the board is fine. Listen to the buzzer to confirm flight state in that case.

Techical Aspects

Sensor Data Path

It's a tricky situation trying to fetch data from all sensors at the exact same time and passing them on to the state machine, the state machine filter and the data logger. Here is a rough overview of how AURORA's data is passed between threads to the state machine and the data logger:


Auxspace e.V. - AURORA
Sensor Data Path
data_logger_init()
baro_probe()
gyro_probe()
state_machine_init()
1
2
3
4
5
6
get_baro()
get_gyro()
zbus_sub_wait_msg()
[baro_ready]
[gyro_ready]
read
no
yes
data_logger_log()
[received inital data from all sensors ?]
baro_submit()
gyro_submit()
state_machine_update()
[STATE_LANDED]
write
ZBus

Threads

`sensor_board` uses up to three additional Zephyr threads (all priority 5, 2 KB stack):

Thread	Guard	Purpose
`imu_task`	`CONFIG_IMU`	Polls IMU at the configured frequency, updates orientation and acceleration globals.
`baro_task`	`CONFIG_BARO`	Measures pressure/temperature, computes altitude.
`state_machine_task`	`CONFIG_AURORA_STATE_MACHINE`	Runs at 10 Hz. Feeds sensor data into the state machine and fires pyro channels on state transitions.

When sensors are configured to use active polling, `baro_task` and `imu_task` run the whole lifetime of the application. When baro, imu or both are using interrupt triggers, `baro_task` and `imu_task` complete after initialisation and the generic sensor trigger threads are running.

> **Note**
>
> To get an overview of running threads, use the Zephyr-Shell builtin command `kernel threads list`.

Configuration

State Machine Thresholds

These options are defined in `sensor_board/Kconfig` under the **State Machine Configuration** menu.

Flight Thresholds

Option	Unit	Default	Description
`CONFIG_BOOST_ACCELERATION`	m/s²	30	Acceleration threshold for boost detection (ARMED -> BOOST).
`CONFIG_BOOST_ALTITUDE`	m	50	Altitude threshold for boost detection (ARMED -> BOOST).
`CONFIG_BURNOUT_ACCELERATION`	m/s²	15	Acceleration threshold for burnout detection (BOOST -> BURNOUT).
`CONFIG_MAIN_DESCENT_HEIGHT`	m	200	Descent altitude for main pyro event (APOGEE -> MAIN).
`CONFIG_LANDING_VELOCITY`	m/s	2	Velocity threshold for landing detection.
`CONFIG_ARM_ANGLE`	deg	85	Orientation threshold for arming (IDLE -> ARMED). 0 = horizontal, 90 = vertical.
`CONFIG_DISARM_ANGLE`	deg	70	Orientation threshold for disarming (ARMED -> IDLE).
`DISARM_ANGLE_SAMPLES`	n	10	Number of consecutive samples for disarming (ARMED -> IDLE).

Timers and Timeouts

Option	Unit	Default	Description
`CONFIG_BOOST_TIMER_MS`	ms	900	Duration that boost thresholds must be held before transitioning.

Option	Unit	Default	Description
`CONFIG_LANDING_TIMER_MS`	ms	500	Duration that landing velocity must be held.
`CONFIG_APOGEE_TIMEOUT_MS`	ms	60000	Max time in APOGEE state before aborting to ERROR.
`CONFIG_MAIN_TIMEOUT_MS`	ms	2000	Delay between MAIN and REDUNDANT pyro events.
`CONFIG_REDUNDANT_TIMEOUT_MS`	ms	900000	Max time in REDUNDANT state before aborting.

Application Simulation

When built with `CONFIG_AURORA_FAKE_SENSORS=y` (typically together with the `native_sim` board target), `sensor_board` replaces the real IMU and baro polling threads with a synthetic data source. The fake threads publish on the same zbus channels (`imu_data_chan`, `baro_data_chan`) at the same cadence as the real drivers, so the state machine, filter, data logger and pyro logic run unchanged.

Two backends sit behind the same shell interface:

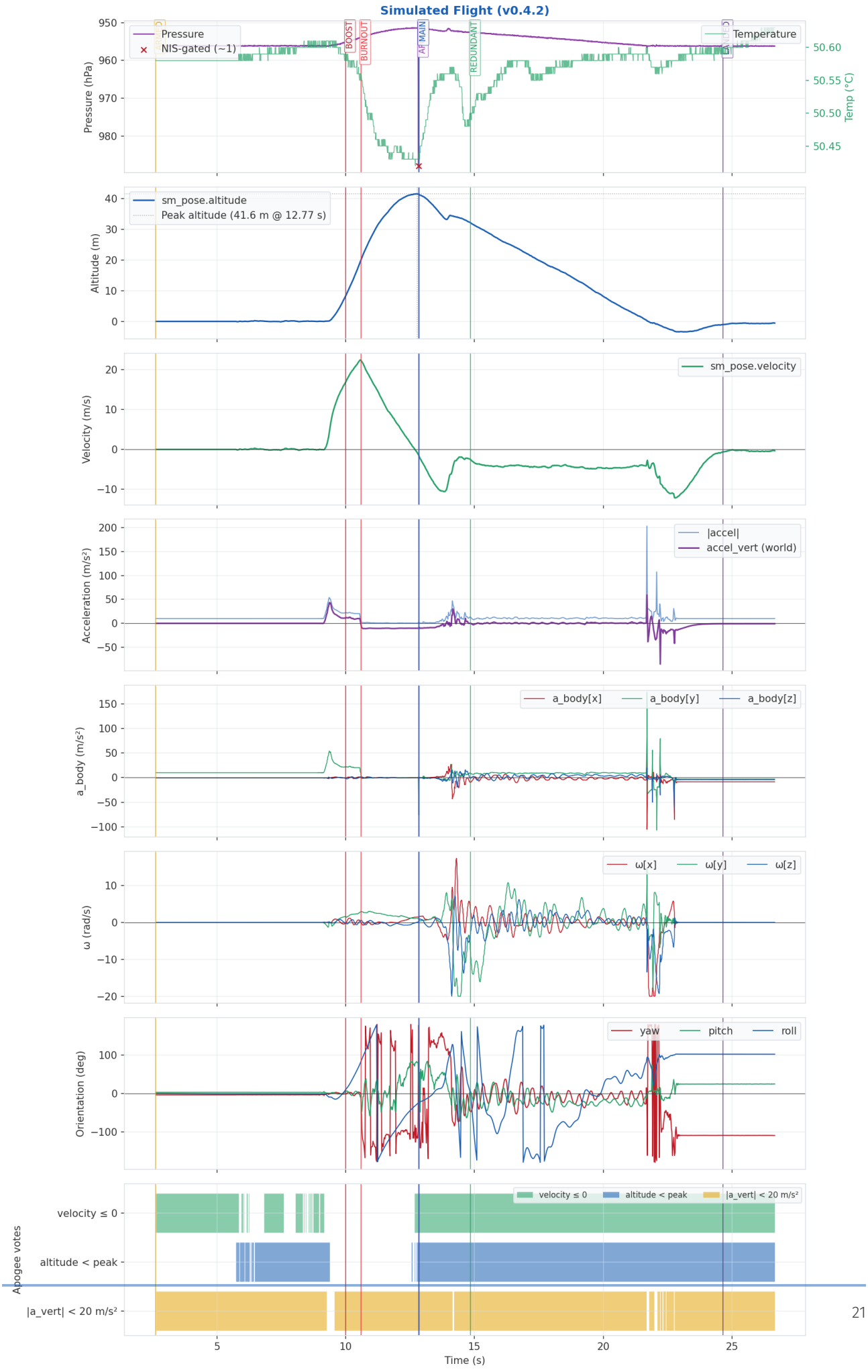
Backend	Kconfig	Source of samples
Synthetic profile	`AURORA_FAKE_SENSORS_SYNTH` (default)	Analytic ISA-troposphere flight profile generated on the fly.
Replay	`AURORA_FAKE_SENSORS_REPLAY`	Samples from a recorded `flights.csv`, embedded in the firmware at build time.

Synthetic profile

The default profile follows an ISA-troposphere altitude/pressure curve with a constant-thrust boost phase, a ballistic coast to apogee and a constant-rate parachute descent. It is controlled from the Zephyr shell via the `sim` command group:

Command	Description
`sim launch`	Start the synthetic flight profile. The uptime at which this command is issued is used as t=0.
`sim reset`	Return the profile to pad-stationary (altitude 0, accel = +g on the vertical axis).
`sim status`	Print the current flight time, altitude (m) and vertical proper-acceleration (m/s²), or `pad-stationary` if no launch is active.

Using the `native_sim` board target, and the `sim_fetch.py` tool, it is possible to run application simulations and create graphs from the data_logger output:


Simulated Flight (v0.4.2)
Pressure
NIS-gated (~1)
Temperature
BOOST
BURNOUT
APOGEE
MAIN
REDUNDANT
LANDED
Pressure (hPa)
Temp (°C)
sm_pose.altitude
Peak altitude (41.6 m @ 12.77 s)
Altitude (m)
sm_pose.velocity
Velocity (m/s)
|accel|
accel_vert (world)
Acceleration (m/s²)
a_body[x]
a_body[y]
a_body[z]
a_body (m/s²)
ω[x]
ω[y]
ω[z]
ω (rad/s)
yaw
pitch
roll
Orientation (deg)
velocity ≤ 0
altitude < peak
|a_vert| < 20 m/s²
Apogee votes
Time (s)

Replay backend

Setting `CONFIG_AURORA_FAKE_SENSORS_REPLAY=y` swaps the analytic profile for a playback engine that streams a previously recorded flight back into the system at the original cadence. The samples are baked into the firmware image at build time, so no filesystem or network access is needed at runtime. Useful for CI runs and for regression-testing the Kalman filter / state machine against known-good data.

The recording is selected via:

```
CONFIG_AURORA_FAKE_SENSORS_REPLAY_INPUT="flight_logs/<campaign>/
<flight>/flights.csv"
```

(Paths are relative to the aurora module root.) During the build, the `gen_flight_replay.py` tool turns the CSV, and (if a sibling `state_audit` file is present) the trimmed `[BOOST - 4 s, LANDED + 4 s]` window, into a generated `replay_data.c` that is linked into the firmware.

Before launch, the replay threads keep the rocket "pad-stationary" by republishing the very first recorded sample, so attitude calibration converges as it would on the real hardware. Issuing `sim launch` from the shell (or letting `CONFIG_AURORA_SIM_AUTOTEST=y` do it automatically) then starts the playback.

Supported Boards and Shields

Since `sensor_board` is an auxspace internal project, only auxspace hardware is tested with the application.

- Sensor Board v2 - RP2040 / RP2350 flight computer
- ESP32-S3 Micrometer - ESP32-S3 based board

Hardware Requirements

Any board that wants to run `sensor_board` has to provide the following peripherals and wire them up via Zephyr `chosen` nodes. The chosen-node names below are what the application looks up at boot. If a node is missing the corresponding feature is silently disabled (or the build fails, in the case of the sensors).

Requirement	Chosen node	Notes
IMU	`auxspace,imu`	6-DoF accelerometer + gyro. Needs at least >=100 Hz ODR to catch boost cleanly. Tested with LSM6DSO32.

Requirement	Chosen node	Notes
Barometric sensor	`auxspace,baro`	Absolute pressure sensor for altitude. Tested with MS5607 and LPS22HH.
Storage device	`auxspace,mmc`	µSD-Card or eMMC, **at least 16 GiB**. Flight computers often have no on-board storage, so an external card is required.
FAT filesystem	`auxspace,ffs`	A `zephyr,fstab,fatfs` entry on top of the storage device, used for human-readable artefacts (configs, exported logs).
Flight-log raw region	`auxspace,flight-log-disk`	A reserved raw region on the same storage device used by the data logger for high-rate writes. Needs ≥7 GiB (see the storage-access pattern caveats).
Pyro driver	`auxspace,pyro`	At least two pyro channels (drogue/main) plus a redundant channel if you want the `REDUNDANT` state to do anything.
Battery	-	Sized for at least the longest realistic flight + recovery window. The `LANDED` recovery beacon will keep the buzzer playing until the battery is empty or the board is powered off.
PWM buzzer (optional)	`auxspace,buzzer`	Passive buzzer driven via PWM. Used for boot, calibration, state-change and recovery cues. Optional - board still flies without it, but you lose the auditory cues.
Status LED(s) (optional)	`auxspace,led`	One or more LEDs as children of a `pwm-leds` node. Driven in lockstep. Optional - board still flies without it, but you lose the visual cues.
	`auxspace,pfm`	

Requirement	Chosen node	Notes
Arm/disarm input (optional)		A GPIO-backed switch, button or magnetic key that asserts the ARM signal. Mechanism is up to the board: the application only sees an edge. Optional - not implementing this feature might cause problems when the board is only armed and disarmed via orientation.
Fieldbus (optional)	-	CAN, \(I^2C\) or similar, if the board is part of a multi-PCB AURORA stack. Not required for a standalone flight.

⚠ Warning

µSD cards that do not contain a flight log will be formatted by the firmware. All previous data will be lost.

Zephyr Integration

AURORA is registered as a module via `zephyr/module.yml`. This makes AURORA headers available as `#include <aurora/...>` and exposes custom board definitions and drivers to the Zephyr build system.

For general Zephyr usage, see the Getting Started Guide guide.

Drivers Reference

Pyro Drivers

The pyrotechnic ignition driver class provides a hardware-agnostic API for arming, triggering, and sensing pyro channels. There is also a `pyro-shell` integration for interactive pyro testing.

Configuration

Option	Default	Description
`CONFIG_PYRO_INIT_PRIORITY`	80	Kernel init priority for pyro driver instances.
`PYRO_SHELL`	n	Activate the shell integration for pyro drivers.

Shell Commands

Enabling `CONFIG_PYRO_SHELL` registers the `pyro` command group for interactive bring-up, bench testing and ground-checkouts of pyro channels. All channel commands take a devicetree device name and a numeric channel index.

Command	Description
`pyro devices`	List all ready pyro devices and their channel counts.
`pyro channels <device>`	List the channel indices of `<device>`.
`pyro state <device>`	For every channel, print capacitor voltage and sense voltage (in mV).
`pyro arm <device> <channel>`	Arm a pyro channel.
`pyro disarm <device> <channel>`	Disarm a pyro channel.
`pyro trigger <device> <channel>`	Fire a pyro channel. The channel must be armed.
`pyro secure <device> <channel>`	Secure (short) a pyro channel to safely dissipate stored charge.

Command	Description
`pyro sense <device> <channel>`	Read the sense-ADC of a channel (mV).
`pyro cap <device> <channel>`	Read the firing-capacitor voltage of a channel (mV).

⚠ Warning

`pyro trigger` will fire a live pyrotechnic channel. Only use this command with igniters disconnected or in an explicitly safe test configuration.

API-Reference

Operations

int pyro_arm(const struct device *dev, uint32_t channel)

Arm a pyro channel.

Parameters:

- **dev** – Pyro device instance.
- **channel** – Number of the channel to arm.

Return values:

- **0** – if successful.
- **-errno** – Other negative errno code on failure.

int pyro_disarm(const struct device *dev, uint32_t channel)

Disarm a pyro channel.

Parameters:

- **dev** – Pyro device instance.

- **channel** – Number of the channel to disarm.

Return values:

- **0** – if successful.
- **-errno** – Other negative errno code on failure.

int pyro_secure(const struct device *dev, uint32_t channel)

Secure a pyro channel.

Parameters:

- **dev** – Pyro device instance.
- **channel** – Number of the channel to secure.

Return values:

- **0** – if successful.
- **-errno** – Other negative errno code on failure.

int pyro_trigger_channel(const struct device *dev, uint32_t channel)

Trigger a pyro channel.

Parameters:

- **dev** – Pyro device instance.
- **channel** – Number of the channel to trigger.

Return values:

- **0** – if successful.
- **-EINVAL** – if `channel` can not be set.
- **-errno** – Other negative errno code on failure.

int pyro_charge_channel(const struct device *dev, uint32_t channel)

Charge a pyro channel's capacitor.

Parameters:

- **dev** – Pyro device instance.
- **channel** – Number of the channel to charge.

Return values:

- **0** – if successful.
- **-EINVAL** – if `channel` can not be set.
- **-errno** – Other negative errno code on failure.

int pyro_sense_channel(const struct device *dev, uint32_t channel, uint32_t *val)

Read a pyro channel's "health".

Parameters:

- **dev** – Pyro device instance.
- **channel** – Number of the channel to read the "sense".
- **val** – **[out]** Read value.

Return values:

- **0** – if successful.
- **-EINVAL** – if `channel` is invalid.
- **-errno** – Other negative errno code on failure.

int pyro_read_cap_channel(const struct device *dev, uint32_t channel, uint32_t *val)

Read a pyro channel's capacitor charge.

Parameters:

- **dev** – Pyro device instance.
- **channel** – Number of the channel to read the capacitor voltage of.
- **val** – **[out]** Read value.

Return values:

- **0** – if successful.
- **-EINVAL** – if `channel` is invalid.
- **-errno** – Other negative errno code on failure.

int pyro_get_nchannels(const struct device *dev)

Get the number of pyro channels on this module.

Parameters:

dev – Pyro device instance.

Return values:

-errno – Negative errno code on failure.

Returns:

Number of channels on success.

PYRO_DEVICE_DT_DEFINE(node_id, init_fn, pm_device, data_ptr, cfg_ptr, level, prio, api_ptr, ...)

Like DEVICE_DT_DEFINE() with pyro specifics.

Defines a device which implements the pyro API.

Parameters:

- **node_id** – The devicetree node identifier.
- **init_fn** – Name of the init function of the driver.
- **pm_device** – PM device resources reference (NULL if device does not use PM).
- **data_ptr** – Pointer to the device's private data.
- **cfg_ptr** – The address to the structure containing the configuration information for this instance of the driver.
- **level** – The initialization level. See SYS_INIT() for details.
- **prio** – Priority within the selected initialization level. See SYS_INIT() for details.
- **api_ptr** – Provides an initial pointer to the API function struct used by the driver. Can be NULL.

PYRO_DEVICE_DT_INST_DEFINE(inst, ...)

Like PYRO_DEVICE_DT_DEFINE() for an instance of a DT_DRV_COMPAT compatible.

Parameters:

- **inst** – instance number. This is replaced by `DT_DRV_COMPAT(inst)` in the call to PYRO_DEVICE_DT_DEFINE().
- **...** – other parameters as expected by PYRO_DEVICE_DT_DEFINE().

struct pyro_driver_api

#include <pyro.h>

Pyro driver class operations.

Library Reference

Data Logger

Format-agnostic telemetry data logger for flight/sensor applications. Sensor readings are captured as `datapoint` and serialised through a pluggable formatter backend (vtable pattern).

Built-in Formatters

The flight-time path uses a fixed-size **binary** formatter that writes directly to a raw flash partition or to a raw region of a disk-access block device. Two text formatters are provided as post-flight conversion targets.

- **Binary** (`CONFIG_DATA_LOGGER_BIN`): live flight-time log, no filesystem. Two backends are selectable via `CONFIG_DATA_LOGGER_BIN_BACKEND`:
  - `DATA_LOGGER_BIN_BACKEND_FLASH` — internal flash partition, treated as a circular ring with **boost-freeze** semantics.
  - `DATA_LOGGER_BIN_BACKEND_DISK` — sector range of a Zephyr disk-access device (typically SD card), written linearly through a RAM ring buffer.

  See Binary Flight-Time Log below.
- **CSV** (`CONFIG_DATA_LOGGER_CONVERT_CSV`): conversion target. One header row, then one row per `CONFIG_DATA_LOGGER_CSV_WINDOW_NS` time window. Each row carries one full sensor snapshot — every datapoint whose timestamp falls within the window is merged into the same row. Cells for sensor groups that did not produce a sample in a given window are left empty.
- **InfluxDB Line Protocol** (`CONFIG_DATA_LOGGER_CONVERT_INFLUX`): conversion target. One line per datapoint, no header row.

Binary Flight-Time Log

The binary formatter is the live writer used during a flight. It bypasses the filesystem entirely and writes frame-aligned blocks straight to one of two backends. The on-storage layout is identical across backends: a sequence of fixed-size *frames* (`CONFIG_DATA_LOGGER_BIN_FRAME_SIZE`, typically one flash erase block of 4096 bytes). Each frame begins with a 32-byte `aurora_bin_frame_header` followed by densely packed 32-byte `aurora_bin_record` entries; unused bytes at the tail of a partial frame remain in the erased (`0xFF`) state. Records preserve the `sensor_value` channels losslessly so post-flight tooling can replay filters and the state machine bit-exactly.

Flash Backend (circular)

Targets a fixed flash partition selected via the device-tree chosen entry `auxspace,flight-log`:

```
&flash0 {
    partitions {
        compatible = "fixed-partitions";
        #address-cells = <1>;
        #size-cells  = <1>;
        flight_log: partition@300000 {
            label = "flight_log";
            reg   = <0x300000 DT_SIZE_M(1)>;
        };
    };
};
/ { chosen { auxspace,flight-log = &flight_log; }; };
```

The partition is treated as a **circular ring**. Pre-boost telemetry overwrites old data continuously; once a `DLE_BOOST` event is delivered via `data_logger_event()`, the ring is frozen forward from that point and writes proceed linearly until the partition fills or `DLE_LANDED` plus the post-landed pad window closes the logger. This trades unbounded pre-boost history for a guaranteed BOOST→LANDED window on partitions too small to hold a full flight.

Producer-side records are memcpy'd into DMA-aligned RAM staging buffers; `CONFIG_DATA_LOGGER_BIN_BUF_COUNT` (default 3, triple buffering) absorbs the per-sector erase latency (~25 ms on QSPI NOR). The writer thread issues one `flash_area_erase()` plus one `flash_area_write()` per frame.

Disk Backend (linear ring buffer)

Targets a sector range of a Zephyr disk-access device (typically an SD card via `zephyr,sdmmc-disk`). The region is described by an `auxspaceev,flight-log-disk` node referenced through the chosen entry `auxspace,flight-log-disk`:

```
/ {
    chosen {
        auxspace,flight-log-disk = &flight_log_disk;
    };

    flight_log_disk: flight-log-disk {
        compatible = "auxspaceev,flight-log-disk";
        disk-name = "MMC";
        offset-bytes = <0x40000000>;  /* skip first 1 GiB of card */
        size-bytes   = <0x20000000>;  /* 512 MiB region */
    };
};
```

Both `offset-bytes` and `size-bytes` must be whole multiples of the disk's logical sector size (queried at runtime via `DISK_IOCTL_GET_SECTOR_SIZE` — always 512 on SD/MMC), and `size-bytes` must additionally be a whole multiple of `CONFIG_DATA_LOGGER_BIN_FRAME_SIZE`. The DT cell width

caps either value at 4 GiB - 1. The application is responsible for ensuring any filesystem on the same card does not overlap this range.

Auto-formatting the companion FAT volume

`CONFIG_DATA_LOGGER_DISK_AUTO_MKFS` enables a guarded boot-time `fs_mkfs()` of the FAT volume that shares the SD card with the raw flight-log region. After Zephyr's fstab has tried to automount the volume, the first sector of the flight-log raw region is read and tested for the `AURORA_BIN_FRAME_MAGIC`:

- If a flight frame is present, the card is left untouched — recovery of an existing flight log takes priority.
- Otherwise the FAT volume is unmounted (if fstab automounted it), reformatted, and remounted, even if it was previously healthy.

Flight-log magic is the sole gate, so a card carrying a valid FAT but no flight log is reformatted on the next boot. This is what makes a fresh SD card and a sim run with a blank RAM disk both come up with a usable filesystem without manual intervention, while still guaranteeing that an existing flight log is never overwritten. Requires the DT chosen `auxspace,ffs` to point at the `zephyr,fstab,fatfs` entry whose disk-name matches the flight-log-disk node.

Both arms of the guard are exercised by the `aurora.lib.data.disk_auto_mkfs` ztest suite under `aurora/tests/lib/data_disk_auto_mkfs`. The suite runs on `native_sim` against a RAM disk:

- `disk_auto_mkfs_blank` boots from a blank RAM disk, lets the `SYS_INIT` hook reformat and remount the FAT volume, and asserts that ordinary file I/O round-trips correctly.
- `disk_auto_mkfs_preserve` seeds the configured raw-region offset with `AURORA_BIN_FRAME_MAGIC`, re-invokes the auto-format entry point to mimic a reboot from a populated card, and asserts that the magic survives the call and the FAT volume remains mounted.

The disk writer is purely **linear** from the configured offset — the region is sized for many minutes of flight, so circular wrap is not used. `DLE_BOOST` is recorded but does not freeze a ring.

Producer records are memcpy'd into a contiguous in-RAM ring of `CONFIG_DATA_LOGGER_BIN_RING_FRAMES` slots (must be a power of two; effective capacity is `RING_FRAMES - 1` committed frames since one slot is always reserved for the producer). The writer thread coalesces up to `CONFIG_DATA_LOGGER_BIN_MAX_BATCH_FRAMES` contiguous committed frames into a single `disk_access_write()` call to amortise SD-stack overhead and play to the card's preference for multi-block transfers. At the default 16 slots × 4 KiB frames the ring buffers ~64 KiB of telemetry, enough to ride out an SD card's 100+ ms internal garbage-collection stall before the producer back-pressures.

Common Behaviour

If the producer cannot get a free buffer/slot within `CONFIG_DATA_LOGGER_BIN_PRODUCER_TIMEOUT_MS` it drops records and sets a sticky error on the logger context.

Only one binary logger instance can be live at a time; `data_logger_init()` rejects a second concurrent open of the bin formatter.

Per-Flight Framing

Every frame header carries a `flight_id` (the high-resolution uptime clock at the moment the logger was opened) and a monotonic `seq` starting at 0. The converter walks frames from offset 0 and stops at the first frame whose magic is invalid (== unwritten storage) **or** whose `flight_id` differs from the first frame's — that is how the boundary between the current flight and any leftover data from a previous flight is detected. The storage region is **not** erased between flights; old data is simply ignored.

Lifecycle Events

The upstream application drives flight transitions into the formatter via `data_logger_event()`:

- `DLE_BOOST` — boost detected. The flash backend freezes its ring forward from this point; the disk backend records the event without behavioural change.
- `DLE_LANDED` — landed. The upstream caller keeps the logger open for `CONFIG_DATA_LOGGER_BIN_POST_LANDED_PAD_MS` to capture post-landed telemetry, then closes it.

Combined with pre-boost data already captured by the circular ring on the flash backend, the converted log spans roughly [BOOST minus whatever pre-boost padding fits in the ring, LANDED + post-landed pad].

Post-Flight Conversion

After landing, `data_logger_convert()` replays the binary log through any text formatter:

```
/* Convert the on-storage log to CSV on the filesystem. */
data_logger_convert(&data_logger_csv_formatter, "/data/flight.csv");
```

The flight-log region is left intact. Conversion must not run concurrently with active logging.

Example Usage

```
static struct data_logger logger;

data_logger_init(&logger, "flight", &data_logger_bin_formatter);
data_logger_set_default(&logger);
data_logger_start(&logger);

struct datapoint dp = {
    .timestamp_ns = k_ticks_to_ns_floor64(k_uptime_ticks()),
    .type         = AURORA_DATA_BARO,
    .channel_count = 2,
    .channels = {
        { .val1 = 23, .val2 = 500000 },  /* temperature: 23.5 °C */
        { .val1 = 101325, .val2 = 0 },   /* pressure: 101325 Pa  */
    },
};

data_logger_log(&dp);                          /* default logger  */
data_logger_event(&logger, DLE_BOOST);         /* state-machine   */
/* ... */
data_logger_event(&logger, DLE_LANDED);
data_logger_flush(&logger);
data_logger_close(&logger);
```

Shell Commands

Enabling `CONFIG_DATA_LOGGER_SHELL` registers the `data_logger` command group. All commands operate on loggers registered through `data_logger_init()` logger names tab-complete.

Command	Description
`data_logger list`	List every registered logger with its formatter and current state (`running`, `stopped` or `closed`).
`data_logger start <name>`	Start a logger. Writes the formatter header if applicable.
`data_logger stop <name>`	Stop a logger. Subsequent writes are dropped until restarted.
	Show the state of a single logger.

Command	Description
`data_logger status <name>`	
`data_logger flush <name>`	Flush buffered data to the backing storage.

API Reference

enum aurora_data

Sensor group identifier.

Always add new entries **before** `AURORA_DATA_COUNT` so that the count stays accurate and array-based tables remain valid.

Values:

enumerator AURORA_DATA_BARO

Barometer: [0] temperature, [1] pressure

enumerator AURORA_DATA_IMU_ACCEL

Accelerometer: [0] x, [1] y, [2] z

enumerator AURORA_DATA_IMU_GYRO

Gyroscope: [0] x, [1] y, [2] z

enumerator AURORA_DATA_IMU_MAG

Magnetometer: [0] x, [1] y, [2] z

enumerator AURORA_DATA_SM_KINEMATICS

SM inputs: [0] accel, [1] accel_vert

enumerator AURORA_DATA_SM_POSE

SM Pose: [0] velocity, [1] altitude

enumerator AURORA_DATA_ORIENTATION

Orientation: [0] yaw, [1] pitch, [2] roll

enumerator AURORA_DATA_VBAT

Battery: [0] voltage

enumerator AURORA_DATA_COUNT

Sentinel — do not use as a type

enum data_logger_event

Lifecycle events the upstream application can deliver to a formatter via data_logger_event.

The Binary log format formatter uses these to switch from circular pre-boost capture to linear “freeze” mode at BOOST and to mark the end of the recorded flight at LANDED. Other formatters may ignore them.

Values:

enumerator DLE_BOOST

Boost detected; freeze the ring forward from here.

enumerator DLE_LANDED

Landed; the post-landed pad window starts now.

typedef void (*data_logger_cb_t)(struct data_logger *logger, void *user_data)

Callback type for data_logger_foreach.

Param logger:

Registered logger instance.

Param user_data:

Opaque context passed through from the caller.

int data_logger_init(struct data_logger *logger, const char *filename, const struct data_logger_formatter *fmt)

Initialise a logger.

The output file is placed at `CONFIG_DATA_LOGGER_BASE_PATH/`[`filename`]_N.file_ext, where N is a free rotation index and file_ext comes from `fmt`. Calls `fmt->init` then `fmt->write_header`. On any failure the logger is left in an invalid state and should not be used.

Parameters:

- **logger** – Caller-allocated logger instance.
- **filename** – Base filename (without extension).
- **fmt** – Formatter vtable (e.g. `data_logger_bin_formatter`).

Return values:

0 – on success, negative errno on failure.

bool flight_log_online(void)

Report whether the raw flight-log disk came up at boot.

Latched once by the flight-log-disk SYS_INIT bring-up (see flight_log_disk_auto_format.c). Consumed by the flight state machine as an arming precondition: the vehicle must not arm and therefore must not fly or fire pyros, if it cannot record the flight. Only defined when `CONFIG_DATA_LOGGER_DISK_AUTO_MKFS` is enabled.

Return values:

- **true** – The flight-log disk is reachable (disk_access OK at boot).
- **false** – The disk failed to initialise; arming should be inhibited.

void data_logger_set_default(struct data_logger *logger)

Set the default logger used by data_logger_log.

Parameters:

logger – Initialised logger instance (NULL to clear).

int data_logger_log(const struct datapoint *dp)

Log a datapoint to the default logger.

Uses the logger previously registered with data_logger_set_default. Returns -ENODEV if no default logger has been set.

Parameters:

dp – Datapoint to write.

Return values:

0 – on success, negative errno on failure.

int data_logger_write(struct data_logger *logger, const struct datapoint *dp)

Serialise and store one datapoint to a specific logger.

Parameters:

- **logger** – Initialised logger instance.

· **dp** – Datapoint to write.

Return values:

0 – on success, negative errno on failure.

int data_logger_flush(struct data_logger *logger)

Flush buffered data to the underlying storage.

Parameters:

logger – Initialised logger instance.

Return values:

0 – on success, negative errno on failure.

int data_logger_close(struct data_logger *logger)

Finalise the output file and release formatter resources.

After this call `logger` is reset and must be re-initialised before use.

Parameters:

logger – Initialised logger instance.

Return values:

0 – on success, negative errno on failure.

int data_logger_stop(struct data_logger *logger)

Temporary stop the data logger and flush the fs caches.

After this call `logger` is stopped and must be re-started before use.

Parameters:

logger – Initialised logger instance.

Return values:

0 – on success, negative errno on failure.

int data_logger_start(struct data_logger *logger)

Restart the data logger.

After this call `logger` is started and running.

Parameters:

logger – Initialized but stopped logger instance.

Return values:

0 – on success, negative errno on failure.

int data_logger_event(struct data_logger *logger, enum data_logger_event ev)

Deliver a flight-lifecycle event to the formatter.

Dispatches to the formatter's optional `on_event` hook under the logger mutex. Formatters that do not implement the hook silently accept the event.

Parameters:

- **logger** – Initialised logger instance.
- **ev** – Event to deliver.

Return values:

0 – on success or no-op, negative errno on failure.

const char *data_logger_type_name(enum aurora_data type)

Return the human-readable name for an aurora_data value.

Parameters:

type – Sensor group identifier.

Returns:

Null-terminated ASCII string, or `"unknown"` for out-of-range values.

void data_logger_foreach(data_logger_cb_t cb, void *user_data)

Iterate over all registered data loggers.

Parameters:

- **cb** – Callback invoked for each registered logger.
- **user_data** – Opaque pointer forwarded to `cb`.

struct data_logger *data_logger_get(const char *name)

Look up a registered data logger by name.

Parameters:

name – Logger name (as passed to data_logger_init).

Returns:

Pointer to the logger, or NULL if not found.

DP_MAX_CHANNELS

Maximum number of sensor channels carried by a single datapoint.

DATA_LOGGER_NAME_MAX

Maximum length of a data logger name (including NUL).

DATA_LOGGER_PATH_MAX

Maximum length of a data logger path (including NUL).

struct datapoint

#include <data_logger.h>

Flat telemetry data point.

Carries a timestamp, sensor-group type, the number of valid channels, and up to DP_MAX_CHANNELS raw Zephyr `sensor_value` readings.

struct data_logger_formatter

#include <data_logger.h>

Formatter vtable.

Implement all mandatory callbacks and export a `const struct data_logger_formatter` to create a new backend.

Every callback receives the logger instance so it can reach its opaque context via `logger->ctx`. Return 0 on success, negative errno on failure.

struct data_logger_state

#include <data_logger.h>

Formatter state.

struct data_logger

#include <data_logger.h>

Logger instance (caller-allocated, typically stack or static).

Disk activity LED

A blinking “the storage card is busy” light. The same idea as the little activity LED on a USB stick or an old hard drive. It flickers while data is being written to the card and stays dark when the card is idle.

During a flight the board streams sensor data to the SD card. This module drives a small GPIO LED so a human standing next to the rocket can *see* that logging is actually happening.

It is kept completely separate from the PWM status LED used by the rest of the notification library: that LED shows the **flight state**, this one shows **disk activity**, and they sit on different pins so they can never overwrite each other.

How it works

Whatever code writes to the SD card (the flight data logger) calls `disk_led_activity()` once per write. Each call does two things:

1. turns the LED **on**, and
2. arms a countdown (the “hold timer”) to turn it **off** again a few milliseconds later (`CONFIG_AURORA_NOTIFY_DISK_LED_HOLD_MS`, default 40 ms).

The trick is that every new write *restarts* the countdown. So the timer only ever fires once the card has been quiet for a whole hold window:

What the card is doing	What you see on the LED
a single quick write	one short blink
a rapid burst of writes	a fast flicker
continuous writing	looks steadily on

What the card is doing	What you see on the LED
idle	off

The “turn off later” step runs on a Zephyr *delayable work item* rather than by sleeping, so `disk_led_activity()` returns immediately and never stalls the logger thread that called it.

How to wire it into a board

A board opts in with two things:

1. a devicetree `chosen` entry `auxspace,disk-led` pointing at a `gpio-leds` child node that says which pin the LED is on, for example:

```
/ {
    status_leds: status-leds {
        compatible = "gpio-leds";

        sd_activity_led: led_sd {
            gpios = <&gpio1 7 GPIO_ACTIVE_HIGH>;
            label = "SD Activity LED";
        };
    };

    chosen {
        auxspace,disk-led = &sd_activity_led;
    };
};
```

2. `CONFIG_AURORA_NOTIFY_DISK_LED=y` in the board’s application config.

If either is missing, `disk_led_activity()` compiles down to a do-nothing call, so shared code (such as the data logger) can invoke it unconditionally without `#ifdef`s.

Configuration

Kconfig	Default	Purpose
`CONFIG_AURORA_NOTIFY_DISK_LED`	n	Enable the SD activity LED backend.
`CONFIG_AURORA_NOTIFY_DISK_LED_HOLD_MS`	40	

Kconfig	Default	Purpose
		How long the LED stays lit after the most recent disk access.

API Reference

static inline void disk_led_activity(void)

Report an SD-card access; blinks the activity LED.

Call once per SD write. Lights the LED and re-arms a `CONFIG_AURORA_NOTIFY_DISK_LED_HOLD_MS` off-timer, so frequent calls keep it lit. Returns immediately; safe to call from any thread.

Input Filtering

AURORA is built to use different filter techniques for apogee detection, controlled by `CONFIG_FILTER_TYPE`.

Kalman Filter

One implementation uses a 2-state Kalman filter with vertical acceleration as a control input (guarded by `CONFIG_FILTER` and `CONFIG_FILTER_KALMAN`). The filter tracks a two-element state vector, consisting of altitude and vertical velocity, and is fed barometric altitude measurements that the baro library converts from pressure via the ISA hypsometric formula.

On every state-machine tick the filter runs a predict/update cycle:

- The predict step propagates altitude and velocity forward using the world-frame vertical acceleration as a control input ($B = [\tfrac{1}{2} \Delta t^2,\; \Delta t]^\top$) while growing the covariance by a process-noise term scaled with Δt. Δt is clamped to 1 s to prevent filter blow-up.
- The update step corrects the state with the latest barometric altitude reading through the standard Kalman gain. A normalised-innovation-squared (NIS / Mahalanobis) gate of 25 ($\approx 5\sigma$) rejects obvious sensor glitches; the gate is disabled for the first 30 updates (~3 s at 10 Hz) to ride out the initial transient before the prior covariance settles.

Apogee detection combines three criteria evaluated on every call to `filter_detect_apogee()`:

- velocity estimate has gone non-positive,

- the running peak altitude is no longer being beaten (descent), and
- the last applied vertical acceleration sits inside a \(\pm 20\,\mathrm{m/s^2}\) coast band, which rejects boost-phase kinematics.

All three must hold for `CONFIG_FILTER_APOGEE_DEBOUNCE_SAMPLES` consecutive samples before apogee is latched and the \(\text{BURNOUT} \rightarrow \text{APOGEE}\) state transition is reported. Once latched, the filter keeps reporting apogee without re-evaluating the criteria.

Three tunable noise parameters (altitude process noise Q_alt, velocity process noise Q_vel, and measurement noise R) plus the debounce sample count are exposed as Kconfig options so they can be adjusted without recompiling the filter source.

A Python simulation tool (tools/sim_flight_kalman.py) reproduces the same algorithm with a realistic flight profile and MS5607 sensor-noise model, allowing the filter to be tuned and validated offline before flight. See sim_flight_kalman.py for usage and example output. For plotting recorded telemetry from real flights — including a replay of the NIS gate and the three-vote apogee detector against the logged streams — see plot_flight_data.py.

The filter and its hypsometric pipeline are covered by a ztest suite under aurora/tests/lib/filter/.

Configuration

Noise parameters are integer-scaled by 1000 in Kconfig. The actual float value is `CONFIG_FILTER_*_MILLISCALE / 1000.0`.

Option	Default	Description
`CONFIG_FILTER_Q_ALT_MILLISCALE`	100	Process noise for altitude state (Q_alt). Increase if altitude estimate lags behind reality.
`CONFIG_FILTER_Q_VEL_MILLISCALE`	500	Process noise for velocity state (Q_vel). Increase for more aggressive response during boost.
`CONFIG_FILTER_R_MILLISCALE`	4000	Measurement noise variance (R). Higher values trust the barometer less.
`CONFIG_FILTER_APOGEE_DEBOUNCE_SAMPLES`	3	Consecutive `filter_detect_apogee()` calls for which all apogee criteria (\(v

Option	Default	Description
		\leq 0\), no new peak, coast-band acceleration) must hold before apogee is latched. Not milliscaled.

API-Reference

int filter_init(struct filter *filter)

Initialize the filter.

Zeroes state, sets initial covariance, and loads process/measurement noise from Kconfig (scaled by FILTER_SCALE_DIVISOR).

Parameters:

filter – Pointer to filter structure.

Return values:

- **0** – on success.
- **-EINVAL** – if `filter` is NULL.

int filter_predict(struct filter *filter, int64_t dt, double a_vert)

Perform the filter prediction step.

Propagates state and covariance forward in time using a constant-acceleration model with `a_vert` as the control input.

Parameters:

- **filter** – Pointer to filter structure.
- **dt** – Elapsed time in nanoseconds since last prediction.
- **a_vert** – World-frame vertical acceleration (m/s^2), gravity-removed. Pass 0.0 to fall back to constant-velocity behavior when no acceleration input is available.

Return values:

- **0** – on success.
- **-EINVAL** – if `filter` is NULL or `dt` <= 0.

int filter_update(struct filter *filter, double z)

Perform the filter measurement update step.

Computes Kalman gain and corrects the state estimate using a barometric altitude measurement.

A normalized-innovation-squared (NIS) gate rejects measurements that are statistically inconsistent with the current estimate (y*y/S above ~5-sigma), guarding against sensor glitches such as ejection pressure spikes. The gate only activates once the filter's altitude variance has dropped to or below the measurement noise, so cold-start and wide-prior conditions still accept the first reads.

Parameters:

- **filter** – Pointer to filter structure.
- **z** – Measured altitude in meters.

Return values:

- **0** – if the measurement was applied.
- **1** – if the measurement was rejected by the innovation gate.
- **-EINVAL** – if `filter` is NULL.
- **-EDOM** – if the innovation covariance is singular.

int filter_detect_apogee(struct filter *filter)

Detect apogee using a three-criterion vote.

Combines three signals to reject noise-driven false positives:

1. Filtered vertical velocity is non-positive. Velocity is the leading indicator at apogee, so no hysteresis band is applied beyond the debounce below — it would only delay detection.
2. Filtered altitude is below the tracked peak, guarding against pad-side triggers where velocity jitter could briefly go negative before launch.

3. The most recent world-frame vertical acceleration is within a hard-coded sanity band (|a_vert| < ~20 m/s^2), ruling out boost and anomalous accel transients at the decision point.

All conditions must hold for `CONFIG_FILTER_APOGEE_DEBOUNCE_SAMPLES` consecutive calls before apogee is reported. Apogee is latched – once reported, subsequent calls return 0 until filter_init is called again.

Parameters:

filter – Pointer to filter structure.

Return values:

- **1** – if apogee detected.
- **0** – if apogee not detected.
- **-EINVAL** – if `filter` is NULL.

FILTER_SCALE_DIVISOR

Divisor applied to Kconfig milliscale noise parameters.

struct filter

#include <filter.h>

2-state Kalman filter structure for rocket state machine.

State vector: x[0] = altitude (m) x[1] = vertical velocity (m/s)

The filter uses a constant-acceleration model with vertical acceleration supplied as a control input to filter_predict(), and barometric altitude as the measurement input. Passing a_vert = 0.0 reduces the model to constant-velocity behavior.

Notifications

The notification library provides an abstract interface for user-facing indicators (buzzer, RGB LED, ...). Each backend registers a `notify_backend` at link time via an iterable section. The library fans out every call to all enabled backends.

Backends

- **PWM Buzzer** (`CONFIG_AURORA_NOTIFY_BUZZER`): drives a passive buzzer via PWM to signal boot, calibration start/completion, state changes, and errors. Runs on a dedicated worker thread so that the blocking tone sequences do not stall the caller (typically the state-machine task). See [Buzzer Patterns] and [Threading and Queueing].
- **PWM LED** (`CONFIG_AURORA_NOTIFY_LED`): drives an LED via PWM to signal boot, state changes, and errors. LED does not blink when data logger is disabled. See [LED Patterns].

Notification Patterns

The following tables describe exactly what each backend does for every event in the notification API. Patterns are kept short and distinctive so operators on the launch pad can identify system state by ear and eye alone.

Events

These are the events (hooks) dispatched by the notification library. Every registered backend reacts independently; not every backend reacts to every event.

Event	When it fires
`on_boot`	Once at system startup, after backends are initialised.
`on_calibration_start`	IMU calibration has begun accumulating a stationary window. Raised once per calibration cycle, not on the internal restarts the attitude tracker performs whenever it detects motion.
`on_calibration_complete`	IMU calibration has finished and the rocket is ready for arming.
`on_state_change`	Flight state-machine transition (see state table below).
`on_error`	An unrecoverable error condition was reported.
`on_powerfail`	A power failure was detected, or the system recovered from one.

LED Patterns

The LED backend drives every LED child of the `auxspace_led` chosen node in lockstep (same pattern on all LEDs). Brightness is 100% (`MAX_BRIGHTNESS`) whenever the LED is lit.

Event / State	Pattern	Meaning
Boot	Solid ON for 500 ms, then OFF	System powered up and notification stack initialised.
Calibration started	*(not handled)*	The LED backend does not implement `on_calibration_start`; the `IDLE` blink already covers this phase. Use the buzzer cue.
Calibration complete	Single 50 ms flash	IMU calibration finished, rocket ready to arm.
State → `IDLE`	Blink 50 ms ON / 450 ms OFF (short pulse, ~2 Hz)	Safe, disarmed. IMU bias calibration runs in the background here; pyros are **not** yet live.
State → `ARMED`	Blink 200 ms ON / 200 ms OFF (even, ~2.5 Hz)	Calibrated and awaiting launch detection. **Pyros live.**
State → `LANDED`	Blink 400 ms ON / 100 ms OFF (long pulse, ~2 Hz)	Flight complete, rocket on ground. Safe to recover.
State → `ERROR`	Solid ON	Unrecoverable error. Service required.
Any other state transition	All LEDs OFF	In-flight states (`BOOST`, `BURNOUT`, `APOGEE`, `MAIN`, `REDUNDANT`) are silent on the LED to save power and avoid optical noise during flight.
Powerfail (loss)	All LEDs OFF; subsequent state/	Power dropped below threshold; backend is muted until recovery to conserve the backup rail.

Event / State	Pattern	Meaning
	error events suppressed	
Powerfail (recover)	LEDs resume normal behaviour on the next event	Main power restored.

> **Note**
>
> When a powerfail has been signalled and not yet recovered, the LED backend ignores state changes and error events. It will re-enable itself on the next event after `on_powerfail(recover=1)`.

Buzzer Patterns

The buzzer backend drives a passive PWM buzzer on the `auxspace_buzzer` chosen node. Every state transition first stops any currently playing melody before issuing the new pattern.

Event / State	Pattern	Meaning
Boot	4000 Hz tone for 500 ms	System powered up.
Calibration started	Two 1000 Hz beeps of 100 ms, separated by a 100 ms gap	The stationary IMU calibration window has begun accumulating (during `IDLE`). Same pitch as the “Calibration complete” tone below so both read as “calibration” by ear, with the rhythm distinguishing them (two short = started, one long = done); deliberately low against the 4000 Hz boot jingle that precedes it in the normal power-on flow. Fires once per calibration cycle — the window

Event / State	Pattern	Meaning
		restarts the attitude tracker performs on motion are silent.
Calibration complete	1000 Hz tone for 500 ms	IMU calibration finished (runs in the background during `IDLE`), rocket ready to arm. When arming is requested and calibration is already done, this is immediately followed by the `ARMED` beep below.
State → `IDLE`	500 Hz tone for 50 ms (low, short chirp)	Safe, disarmed.
State → `ARMED`	2000 Hz tone for 200 ms (mid, clear beep — higher/longer than the "Calibration complete" tone above so the two are distinguishable by ear)	Calibrated. **Pyros live.**
State → `APOGEE`	3000 Hz tone for 300 ms (high, longer beep)	Apogee detected, drogue event triggered.
State → `MAIN`	2500 Hz tone for 300 ms (mid-high, longer beep)	Main parachute deployment event.
State → `REDUNDANT`	Two 2500 Hz beeps of 150 ms, separated by a 100 ms gap	Redundant (backup) parachute deployment event. The double beep distinguishes the fallback path from the nominal `MAIN` event.
State → `LANDED`	"Astronomia" (Coffin Dance) melody, plays until interrupted	Flight complete. Acts as an audible recovery beacon.

Event / State	Pattern	Meaning
Any other state transition	Silent (any ongoing melody is stopped)	In-flight states `BOOST` and `BURNOUT` are silent on the buzzer.
Error	Three 4000 Hz beeps of 100 ms, separated by 100 ms gaps	Unrecoverable error. Service required.
Powerfail	*(not handled)*	The buzzer backend does not implement `on_powerfail`.

Note

The `LANDED` melody is a looping recovery beacon and is the only pattern that runs asynchronously; it is stopped automatically on the next state transition.

Quick Reference

State	LED	Buzzer
`IDLE`	Short blink (50 / 450 ms)	500 Hz · 50 ms
`ARMED`	Even blink (200 / 200 ms)	2000 Hz · 200 ms
`BOOST`	Off	Silent
`BURNOUT`	Off	Silent
`APOGEE`	Off	3000 Hz · 300 ms

State	LED	Buzzer
`MAIN`	Off	2500 Hz · 300 ms
`REDUNDANT`	Off	2 × 2500 Hz · 150 ms beeps
`LANDED`	Long blink (400 / 100 ms)	“Astronomia” melody (looping)
`ERROR`	Solid ON	3 × 4000 Hz · 100 ms beeps

Threading and Queueing

The notification dispatcher (`notify_state_change()`, `notify_error()`, …) runs synchronously in the caller’s thread — it fans out to each backend inline. Individual backends may choose to offload their work to avoid blocking flight-critical threads.

Buzzer backend runs on a dedicated worker thread with a bounded FIFO event queue:

- Calls from the state-machine task return immediately (they only enqueue an event), so blocking tone sequences (up to ~600 ms for the error pattern) never stall the 10 Hz state machine.
- Events are played in FIFO order. Important sequencing — notably stopping the `LANDED` melody before a new tone — is preserved because the worker thread runs `pwm_melody_stop` and the subsequent tone in one dequeued step.
- When the queue is full, new events are dropped with a `LOG_WRN`. This gives natural back-pressure: the worker drains at the speed of its tone sequences, and bursty producers cannot unboundedly queue up noise.

Tunables (under `AURORA_NOTIFY_BUZZER`):

Kconfig	Default	Purpose
`AURORA_NOTIFY_BUZZER_QUEUE_SIZE`	16	Maximum queued events before overflow drops.
`AURORA_NOTIFY_BUZZER_STACK_SIZE`	1024	Worker thread stack size (bytes).

Kconfig	Default	Purpose
`AURORA_NOTIFY_BUZZER_THREAD_PRIORITY`	10	Worker thread priority. Keep numerically above flight threads (priority 5) so notifications never preempt them.

LED backend does not need a dedicated thread: blinking is delegated to Zephyr's `pwm-leds` driver (software timer), and the remaining inline sleeps are short (≤ 500 ms at boot, 50 ms on calibration) and occur outside the flight hot path.

API Reference

int notify_init(void)
: Initialise all notification backends.

  Return values:

  0 – on success, or first non-zero return from a backend.

int notify_boot(void)
: Notify all backends that the system has booted.

  Return values:

  0 – on success, or first non-zero return from a backend.

int notify_state_change(enum sm_state prev, enum sm_state next)
: Notify all backends of a state-machine transition.

  Parameters:

  - **prev** – Previous state.
  - **next** – New state.

  Return values:

  0 – on success, or first non-zero return from a backend.

int notify_calibration_start(void)
Notify all backends that IMU calibration has started.

Calibration runs in the background while the state machine is in `SM_IDLE`. The application is expected to raise this once per calibration cycle, when the stationary window actually starts accumulating — not on every internal restart the tracker performs when it detects motion.

Return values:

0 – on success, or first non-zero return from a backend.

int notify_calibration_complete(void)
Notify all backends that IMU calibration has completed and the rocket is ready to launch.

Return values:

0 – on success, or first non-zero return from a backend.

void notify_powerfail(int recover)
Notify all backends about a powerfail.

Parameters:

recover – 1 if Power failure is being recovered.

int notify_error(void)
Notify all backends of an error condition.

Return values:

0 – on success, or first non-zero return from a backend.

NOTIFY_BACKEND_DEFINE(_name, _api)
Register a notification backend (link-time).

Place a notify_backend instance in the iterable section so the notification library discovers it without explicit registration.

Parameters:

- **_name** – Unique C identifier for this backend.
- **_api** – Pointer to a notify_backend_api vtable.

struct notify_backend_api

#include <notify.h>

Notification backend operations.

Each output device (buzzer, RGB LED, ...) implements this vtable.

struct notify_backend

#include <notify.h>

Notification backend descriptor.

Backends are collected automatically at link time via an iterable section; no central registration call is needed.

Pad Link

The pad-link library turns the rocket into a small Bluetooth Low Energy (BLE) peripheral so a nearby ground station (usually a launchrail computer or a laptop running a BLE app) can read the live status while the rocket is sitting on the launch pad.

It is **not** a flight downlink. The BLE radio range is tens of metres at best, and once the motor lights, the rocket will leave that bubble within a second or two. The link is meant for the minutes between `firmware booted` and `launch button pressed`.

If you have used a fitness tracker or a smart watch, the design is the same: the rocket *advertises* (broadcasts "I'm here"), the ground station *scans* and *connects*, and then it reads or subscribes to a handful of *characteristics* (think: named variables on the device).

⚠ Warning

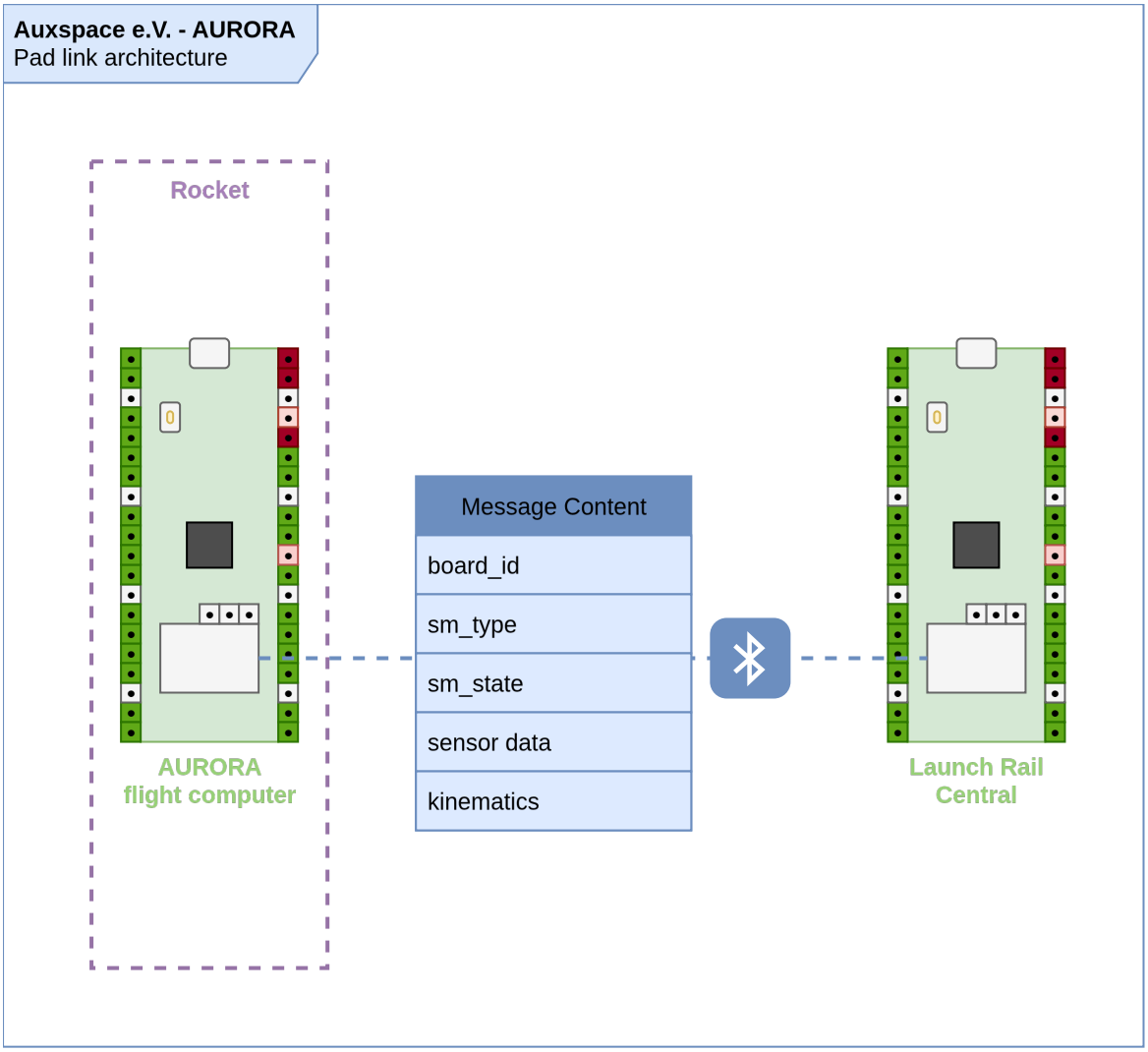
Experimental on ESP32-S3: it can freeze the flight computer.

On the ESP32-S3 boards (`micrometer`), keeping a BLE connection open and streaming live data on the notify path can, after anywhere from a few seconds to a few minutes, make the whole flight computer **freeze**: it stops responding to everything, including the USB console, until you power-cycle it. Once in a while it crashes instead of freezing.

Some causes have already been fixed: a bug where the flight-control loop could get stuck forever waiting on the Bluetooth stack, and a flood of updates sent far faster than any BLE link can actually carry. One freeze still remains, and its root cause is being tracked down on a separate development branch.

Until that is fixed, treat the notify pad link as a **bench and setup convenience, not something to trust during a countdown or flight**:

- Do **not** rely on the notifications while arming or launching. For anything that matters, use polling mode, a USB cable or the Telemetry link instead.
- If a board locks up during testing, power-cycle it. It is almost certainly the pad link, not the code you were working on.
- If you don't need Bluetooth for what you're doing, build with `CONFIG_AURORA_PAD_LINK=n` to leave it out entirely and rule it out as a cause.



When to use it

Use the pad link when you want any of the following:

- Watch the state machine on the pad ("did it really go IDLE -> ARMED when I flipped the arm switch?") without a USB cable.
- Sanity-check the last raw IMU and baro readings before flight.
- Display computed altitude, velocity, and orientation on a phone or laptop during setup.
- Identify which board is on the pad (e.g. `sensor_board_v2/rp2040` vs. an ESP32-S3 variant) when you have several flight computers in a backpack.

Do **not** use it for:

- In-flight telemetry. Use the telemetry library (HC-12 today, more long-range backends later). See Telemetry.
- Sending commands to the rocket. The pad-link is read-only by design. The only way to influence the rocket from the ground is the arm switch and the launch wire.

Connection lifecycle

The rocket is the *peripheral*; the ground station is the *central*. The lifecycle is symmetric and forgiving:

1. `pad_link_init()` brings up the BLE host stack and starts connectable advertising.
2. Any central that scans for the advertised name (default `AURORA-Rocket`) sees the rocket and may connect.
3. While connected, the central reads characteristics directly or subscribes to notifications and is pushed every new value.
4. When the central disconnects, deliberately, or because the rocket just travelled 200 m up in two seconds, the disconnect callback re-starts advertising. No state on the rocket cares whether anyone is listening.

The flight code never waits on the link. A central showing up an hour after boot, or never showing up at all, is the same code path for the rocket.

GATT service

All data is grouped under one custom *primary service*. A service is just a container. The interesting parts are its *characteristics*. You can think of each characteristic as a named, typed register the central can read.

128-bit UUIDs are used throughout. The service UUID lets a central filter scan results to “AURORA rockets only”:

All UUIDs share the base `e8a591xx-7c0e-4b5b-9a4c-1f1b6f7c4d70`; the low byte `xx` identifies the characteristic.

Name	UUID `xx`	Full UUID	Access
Service	`00`	`e8a59100-7c0e-4b5b-9a4c-1f1b6f7c4d70`	-
Board identifier	`01`	`e8a59101-7c0e-4b5b-9a4c-1f1b6f7c4d70`	read
SM state	`02`	`e8a59102-7c0e-4b5b-9a4c-1f1b6f7c4d70`	read, notify
Raw sensors	`03`	`e8a59103-7c0e-4b5b-9a4c-1f1b6f7c4d70`	read, notify *(deprecated)*
Computed kinematics	`04`	`e8a59104-7c0e-4b5b-9a4c-1f1b6f7c4d70`	read, notify
SM type	`05`	`e8a59105-7c0e-4b5b-9a4c-1f1b6f7c4d70`	read
Board capabilities	`a0`	`e8a591a0-7c0e-4b5b-9a4c-1f1b6f7c4d70`	read
Barometer	`a1`	`e8a591a1-7c0e-4b5b-9a4c-1f1b6f7c4d70`	read, notify
Accelerometer	`a2`	`e8a591a2-7c0e-4b5b-9a4c-1f1b6f7c4d70`	read, notify
Gyrometer	`a3`	`e8a591a3-7c0e-4b5b-9a4c-1f1b6f7c4d70`	read, notify
6-DoF IMU	`a4`	`e8a591a4-7c0e-4b5b-9a4c-1f1b6f7c4d70`	read, notify

Name	UUID `xx`	Full UUID	Access
9-DoF IMU	`a5`	`e8a591a5-7c0e-4b5b-9a4c-1f1b6f7c4d70`	planned
GPS/GNSS	`a6`	`e8a591a6-7c0e-4b5b-9a4c-1f1b6f7c4d70`	planned
Inner temperature	`a7`	`e8a591a7-7c0e-4b5b-9a4c-1f1b6f7c4d70`	read, notify
Motor temperature	`a8`	`e8a591a8-7c0e-4b5b-9a4c-1f1b6f7c4d70`	planned
Hull temperature	`a9`	`e8a591a9-7c0e-4b5b-9a4c-1f1b6f7c4d70`	planned

"Planned" characteristics have UUID defines reserved in `pad_link.c` but no GATT table entry yet; their corresponding bits in the boardcap register remain `0` until the sensor source is wired up.

Read vs. notify

Every characteristic supports *read*: the central asks once and gets the current value. The characteristics marked "read, notify" also support *notify*: the central writes "1" to the characteristic's CCC descriptor and then receives a push every time the rocket updates the snapshot. Notifications are cheaper than polling at the same rate and arrive immediately on each update.

Board capabilities

After connecting, read characteristic `a0` (boardcap) once. It is a `uint32_t` little-endian register that describes which sensor characteristics carry valid data on this specific board. The application declares the value during init via `pad_link_set_caps()`, composing it from the `PL_CAP_*` flags in `include/aurora/lib/pad_link.h` — which sensors a board carries is hardware knowledge the library does not guess at.

The register is split into four byte-sized groups:

Byte 0 — IMU group

Bits	Name	Description
`[2:0]`	IMU type	`0` = none, `1` = 6-DoF (accel + gyro), `2` = 9-DoF (accel + gyro + mag)
`[3]`	Accel	1 = accelerometer data valid (characteristic `a2`)
`[4]`	Gyro	1 = gyrometer data valid (characteristic `a3`)
`[7:5]`	—	reserved

Byte 1 — Environmental group

Bits	Name	Description
`[8]`	Baro	1 = barometer data valid (characteristic `a1`)
`[9]`	Inner temp	1 = inner temperature data valid (characteristic `a7`)
`[10]`	Motor temp	1 = motor temperature data valid (characteristic `a8`, planned)
`[11]`	Hull temp	1 = hull temperature data valid (characteristic `a9`, planned)
`[15:12]`	—	reserved

Byte 2 — Positioning group

Bits	Name	Description
`[16]`	GPS/GNSS	1 = GPS data valid (characteristic `a6`, planned)

Bits	Name	Description
[23:17]	—	reserved

Byte 3 — reserved

Bits	Name	Description
[31:24]	—	reserved

The IMU type field uses the `pl_cap_imu_type` C enum defined in `lib/pad_link/pad_link_wire.h` (`PL_CAP_IMU_TYPE_NONE`, `PL_CAP_IMU_TYPE_6DOF`, `PL_CAP_IMU_TYPE_9DOF`). The Python example script mirrors these as `CAP_IMU_TYPE_*` constants.

Characteristics in detail

Board identifier: UTF-8 string, length-bounded by `CONFIG_AURORA_PAD_LINK_BOARD_ID`. Defaults to the Zephyr board name (e.g. `sensor_board_v2_rp2040`). Read once; it doesn't change.

SM type: `uint8_t`. Identifies *which* state machine implementation the firmware is running, so the central knows how to decode the SM-state byte. `0` is the simple 9-state SM (see State Machine); future implementations append new values. Read once after connecting.

SM state: `uint8_t`. Current flight state, as defined by the active SM implementation. With `SM type = 0` (simple), the byte maps to `IDLE, ARMED, BOOST, BURNOUT, APOGEE, MAIN, REDUNDANT, LANDED, ERROR`. This ordinal table is append-only (see the decoder warning in `tools/rec_zephyr.py`), so existing values never move.

Raw sensors: packed, little-endian struct. Most recent samples from the IMU and barometer in their native Zephyr `sensor_value` format (`val1` = whole part, `val2` = micro-fraction).

Offset	Size	Type	Field
0	4	`u32`	`uptime_ms` of the latest contributing sample
4	12	`i32[3]`	`accel.val1` (x, y, z)

Offset	Size	Type	Field
16	12	`i32[3]`	`accel.val2` (x, y, z)
28	12	`i32[3]`	`gyro.val1` (x, y, z)
40	12	`i32[3]`	`gyro.val2` (x, y, z)
52	4	`i32`	`temperature.val1`
56	4	`i32`	`temperature.val2`
60	4	`i32`	`pressure.val1`
64	4	`i32`	`pressure.val2`

To recover a physical value, combine the two halves: `value = val1 + val2 * 1e-6`.

Computed kinematics: packed, little-endian struct. The outputs of the state-machine input pipeline (after Kalman filtering, if enabled). 32-bit floats keep the payload small; the precision is far more than the central needs for a status screen.

Offset	Size	Type	Field
0	4	`u32`	`uptime_ms`
4	4	`f32`	`altitude` (m)
8	4	`f32`	`velocity` (m/s, vertical)
12	4	`f32`	`yaw` (deg)

Offset	Size	Type	Field
16	4	`f32`	`pitch` (deg)
20	4	`f32`	`roll` (deg)
24	4	`f32`	`accel_vert` (m/s², world-frame, gravity-removed)

All sensor characteristics below use *micro-units*: divide the raw `int64_t` value by 1 000 000 to recover the physical quantity.

Board capabilities (`a0`): `uint32_t` LE, 4 bytes. See [Board capabilities] above.

Barometer (`a1`): 20 bytes. Present when `PL_CAP_BARO` is set.

Offset	Size	Type	Field
0	4	`u32`	`uptime_ms`
4	8	`i64`	`temp_us` (µ°C)
12	8	`i64`	`press_us` (µPa)

Accelerometer (`a2`): 28 bytes. Present when `PL_CAP_ACCEL` is set.

Offset	Size	Type	Field
0	4	`u32`	`uptime_ms`
4	24	`i64[3]`	`accel_us` x, y, z (µm/s²)

Gyrometer (`a3`): 28 bytes. Present when `PL_CAP_GYRO` is set.

Offset	Size	Type	Field
0	4	`u32`	`uptime_ms`
4	24	`i64[3]`	`gyro_us` x, y, z (µrad/s)

6-DoF IMU (`a4`): 52 bytes. Present when IMU type ≥ 6-DoF. Combines `a2` and `a3` in one characteristic — subscribe to `a4` *or* `a2` + `a3`, not both.

Offset	Size	Type	Field
0	4	`u32`	`uptime_ms`
4	24	`i64[3]`	`accel_us` x, y, z (µm/s²)
28	24	`i64[3]`	`gyro_us` x, y, z (µrad/s)

Inner temperature (`a7`): 12 bytes. Present when `PL_CAP_TEMP_INNER` is set. Sourced from the barometer sensor's die temperature.

Offset	Size	Type	Field
0	4	`u32`	`uptime_ms`
4	8	`i64`	`temp_us` (µ°C)

Rocket-side integration

For the typical application, the integration is three calls and one Kconfig:

```
/* main.c — one-time init */
#include <aurora/lib/pad_link.h>

int main(void)
{
```

```
    /* ... other init ... */

    /* Declare this board's sensor fit (hardware knowledge the
     * library does not guess at), then bring up the BLE stack.
     */
    pad_link_set_caps(PL_CAP_IMU_TYPE(PL_CAP_IMU_TYPE_6DOF) |
                      PL_CAP_ACCEL | PL_CAP_GYRO |
                      PL_CAP_BARO | PL_CAP_TEMP_INNER);
    (void)pad_link_init();   /* non-fatal if it fails */
    return 0;
}

/* In the state-machine loop, right after sm_update(): */
struct sm_inputs sm_snap;

sm_get_inputs(&sm_snap);
pad_link_publish_sm(sm_get_state(), sm_get_type(), &sm_snap);
```

`pad_link_publish_sm()` updates the state and computed-kinematics characteristics and fires notifications for any subscribed central. The sensor characteristics fill themselves in automatically: the library subscribes to the IMU and baro ZBUS channels and snapshots every published sample.

Kconfig

Symbol	Default	Purpose
`AURORA_PAD_LINK`	n	Enable the library. Pulls in `BT` and `BT_PERIPHERAL`.
`AURORA_PAD_LINK_DEVICE_NAME`	`"AURORA-Rocket"`	Name in the advertising payload: what the central displays when scanning.
`AURORA_PAD_LINK_BOARD_ID`	`"$(BOARD)"`	Value of the board-identifier characteristic. Override per hardware revision.

A board configuration also has to enable a working BLE controller for the chip in question (Bluetooth HCI driver, controller stack, …). Those configs live in the board `.conf` files because they vary per SoC.

Ground-station example

The central side is whatever can speak BLE: a phone running `nRF Connect`, a laptop with `bluetoothctl`, the launchrail's own firmware, or a small Python script. The shape of the code is always the same: scan, connect, discover, then read or subscribe.

A runnable Python reference using Bleak ships in-tree as pad_link_central_example.py. It scans by service UUID, reads the board id, SM type, and boardcap register, prints a capability summary, then subscribes to or polls only the sensor characteristics that are actually present on the board:

```
$ pip install bleak
$ python3 aurora/tools/pad_link_central_example.py
connected to sensor_board_v2_esp32s3 (sm_type=0)
capabilities:
  IMU:        6-DoF (accel + gyro)
  Barometer:  present
  Inner temp: present
  Motor temp: not present
  Hull temp:  not present
  GPS/GNSS:   not present
state: IDLE
baro: temp=22.10°C  press=101325.00 Pa
imu6: a=[x=+0.000, y=+0.000, z=-9.810] m/s²  g=[x=+0.000, y=+0.000,
z=+0.000] rad/s
t=12345  alt=+0.1  v=+0.0  ypr=+0.3/-0.1/+0.0
...
```

The mode is selectable from the command line:

- `--mode notify` (default): subscribe to state, computed kinematics, and any sensor characteristics enabled by boardcap. The rocket pushes a notification on every SM tick. Lowest latency, highest radio traffic.
- `--mode poll --interval 1.0`: skip notifications and `read` the characteristics on a timer. The central drives the cadence; the rocket never pushes. Use this for a low-rate status display where every SM-tick update would be wasted.
- `--duration` caps how long the script stays connected (default 60 s).

Read the file itself for the wire-format decoding. The header docstring lists prerequisites, gotchas, and pointers on extending it into a fuller dashboard.

A few notes for the first time you wire this up:

- BLE characteristic reads are limited by the negotiated *MTU*. With the default 23-byte ATT MTU the raw-sensor (68 B) and computed (28 B) payloads still arrive correctly, because the central transparently issues *long reads*. Negotiating a larger MTU (`BleakClient(..., mtu=247)`) makes notifications and reads a single round-trip and is worth doing.

- The first read of `sm_state` *before* the state-machine has run even once returns 0 (`IDLE`). The values stabilise within milliseconds of boot.
- Reconnecting after a disconnect just works: the rocket re-advertises immediately. Centrals typically remember the address and reconnect on the next scan.

Failure modes

The library is designed so nothing it does can break the flight code:

- `bt_enable()` returns an error (no controller, bad config, …) -> `pad_link_init()` logs and returns the error. `main()` ignores it. The flight loop runs normally; you simply won't see any advertising.
- Notification send fails (queue full, central too slow) -> the failure is silently dropped. The next call to `pad_link_publish_sm()` will try again with fresher data.
- Central disconnects mid-flight -> `disconnected` callback re-arms advertising. The next time a central is in range, it can reconnect. The flight loop is unaffected.

There is intentionally no retry logic, no buffering for offline centrals, and no fail-safe mode. The pad link is allowed to be missing, reconnecting and imperfect. That is what the SD-card log and the HC-12 downlink are for.

API Reference

enum pl_cap_imu_type

Values:

enumerator PL_CAP_IMU_TYPE_NONE
enumerator PL_CAP_IMU_TYPE_6DOF
enumerator PL_CAP_IMU_TYPE_9DOF

int pad_link_init(void)

Bring up the BLE stack and start advertising.

Idempotent within a boot: returns the `bt_enable` error on failure and does not retry. The caller should treat a failure as "no pad link this boot" and continue.

Return values:

- **0** – on success.
- **<0** – propagated from `bt_enable`.

void pad_link_set_caps(uint32_t caps)

Declare which sensor characteristics carry valid data.

Publishes `caps` on the boardcap characteristic (a0). What the board actually carries is hardware knowledge the application has and the library does not, so the application composes the value from the `PL_CAP_*`

flags and calls this once during init (any time before a central connects). Until then the register reads as 0 — “no

capabilities”. Safe to call from any context; never blocks.

Parameters:

caps – Capability register value, see `PL_CAP_*`.

void pad_link_publish_sm(enum sm_state state, enum sm_type type, const struct sm_inputs *inputs)

Publish the latest SM state and computed kinematics.

Updates the read snapshot and, for any subscribed central, fires a GATT notification. Safe to call from the SM thread; never blocks.

`type` lets the central pick the right `sm_state` enum mapping

Parameters:

- **state** – Current flight state.
- **type** – Active state machine implementation ID (see sm_get_type). Constant across a boot.
- **inputs** – Snapshot returned by `sm_get_inputs`.

PL_CAP_IMU_TYPE_MASK

PL_CAP_IMU_TYPE(t)

PL_CAP_ACCEL

PL_CAP_GYRO

PL_CAP_BARO

PL_CAP_TEMP_INNER

PL_CAP_TEMP_MOTOR

PL_CAP_TEMP_HULL

PL_CAP_GPS

Powerfail Mitigation

Flight computers often log data on storage devices, like µSD-Cards, eMMCs or other NAND-flashes. To protect against broken file systems (or even blocks) on powerloss, we have a small powerfail mitigation subsystem to handle power failures in a way that preserves data.

A power failure event is detected by a configured gpio pin. In a devicetree, the setup for a pulled-up GPIO pin would look like this:

```
/ {
   chosen {
      auxspace,pfm = &button0;
   };

   buttons {
      compatible = "gpio-keys";

      button0: button_0 {
         gpios = <&gpio0 0 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>;
         label = "Powerfail Pin";
      };
   };
}
```

The subsystem is named Powerfail Mitigation, but saving and recovering state can also be of use outside of the powerloss scope. Another example of using the Powerfail Signal is to stop data loggers, when the avionic system is in a disarmed state. In this case, the pin can act as an arm signal and start the data logger, when the pin is pulled and vice-versa:

```
#include <aurora/lib/powerfail.h>

static int armed = 0;

static void powerfail_assert()
{
   // Gets invoked, when powerfail pin is asserted
   armed = 0;
}

static void powerfail_deassert()
{
   // Gets invoked, when powerfail pin is deasserted
```

```
    armed = 1;
}

int main(void)
{
    powerfail_setup(&powerfail_assert, &powerfail_deassert);

    // --snip--
    // use "armed" here e.g. in a loop
    // --snip--

    return 0;
}
```

API Reference

typedef void (*powerfail_cb_t)(void)
: Powerfail callback type, invoked from ISR context.

void powerfail_setup(powerfail_cb_t assert_cb, powerfail_cb_t deassert_cb)
: Initialise the powerfail mitigation subsystem.

  Parameters:

  - **assert_cb** – Optional callback invoked in ISR context after emergency state save when power failure is detected. May be NULL.
  - **deassert_cb** – Optional callback invoked in ISR context when power is restored (pin returns to default pullup). May be NULL.

PWM Melodies

A landed rocket needs a way to communicate its position to the searching squad. One simple way of making it easier to find is to play loud buzzer sounds in an endless loop until the recovery team finds it.

But only playing loud noises is boring and annoying in testing. Instead, use the PWM Melody API to play music of your liking:

```
/ {
    buzzer0: buzzer_0 {
```

```
        compatible = "auxspaceev,pwm-buzzer";
        pwms = <&ledc0 0 PWM_MSEC(200) PWM_POLARITY_NORMAL>;
    };
};
```

```
#include <aurora/lib/pwm_melody.h>

// dt node that contains a "pwms" child node
static const struct pwm_dt_spec buzzer =
    PWM_DT_SPEC_GET(DT_NODELABEL(buzzer0));

// play astronomia from aurora/lib/pwm_melody.h
PWM_MELODY_CTX_DEFINE(melody_ctx, &buzzer, astronomia, 1024);

int main()
{
    pwm_melody_start(&melody_ctx);

    // --snip--
    // play as long as needed
    // --snip

    pwm_melody_stop(&melody_ctx);
}
```

API Reference

static const struct pwm_melody_note astronomia[] = {{466, 4}, {587, 4}, {587, 4}, {587, 4}, {587, 4}, {523, 4}, {523, 4}, {523, 4}, {523, 4}, {698, 4}, {698, 4}, {698, 4}, {698, 4}, {784, 4}, {784, 4}, {784, 4}, {784, 4}, {784, 4}, {784, 4}, {784, 4}, {784, 4}, {784, 4}, {784, 4}, {784, 4}, {784, 4}, {523, 4}, {466, 4}, {440, 4}, {349, 4}, {392, 4}, {0, 4}, {392, 4}, {587, 4}, {523, 4}, {0, 4}, {466, 4}, {0, 4}, {440, 4}, {0, 4}, {440, 4}, {440, 4}, {523, 4}, {0, 4}, {466, 4}, {440, 4}, {392, 4}, {0, 4}, {392, 4}, {932, 4}, {880, 4}, {932, 4}, {880, 4}, {932, 4}, {392, 4}, {0, 4}, {392, 4}, {932, 4}, {880, 4}, {932, 4}, {880, 4}, {932, 4}, {392, 4}, {0, 4}, {392, 4}, {587, 4}, {523, 4}, {0, 4}, {466, 4}, {0, 4}, {440, 4}, {0, 4}, {440, 4}, {440, 4}, {523, 4}, {0, 4}, {466, 4}, {440, 4}, {392, 4}, {0, 4}, {392, 4}, {932, 4}, {880, 4}, {932, 4}, {880, 4}, {932, 4}, {392, 4}, {0, 4}, {392, 4}, {932, 4}, {880, 4}, {932, 4}, {880, 4}, {932, 4},}

Astronomia (Coffin Dance) melody. Ideal for post-flight celebration.

int pwm_melody_start(struct pwm_melody_ctx *ctx)

Start playing a melody.

Spawns a thread that iterates over the notes in `ctx` and drives the PWM output accordingly.

Parameters:

ctx – Melody player context (must have been initialised, e.g. via PWM_MELODY_CTX_DEFINE).

Returns:

0 on success, or a negative error code.

void pwm_melody_stop(struct pwm_melody_ctx *ctx)

Stop a melody that is currently playing.

Signals the playback thread to stop and waits for it to terminate. The PWM output is turned off before returning.

Parameters:

ctx – Melody player context.

PWM_MELODY_CTX_DEFINE(_name, _pwm, _notes, _stack_size)

Statically define and initialize a melody player context.

This macro allocates the thread stack and initialises the context struct.

Parameters:

- **_name** – Variable name for the context.
- **_pwm** – Pointer to a `pwm_dt_spec` for the buzzer output.
- **_notes** – Array of pwm_melody_note to play.
- **_stack_size** – Stack size in bytes for the playback thread.

struct pwm_melody_note

#include <pwm_melody.h>

A single note in a melody.

struct pwm_melody_ctx

#include <pwm_melody.h>

Runtime context for a melody player instance.

Sensors

The AURORA sensors library is merely an extension of the zephyr sensor driver api with helper functions, different sampling methods and specific workflows.

IMU

Interface for the inertial measurement unit (e.g. LSM6DSO32). Provides orientation (pitch and roll) and acceleration data.

ZBUS_CHAN_DECLARE(imu_data_chan)

ZBUS channel for IMU data.

int imu_poll(const struct device *dev)

Poll the IMU for acceleration data and sends it over the z-bus.

Parameters:

dev – Pointer to the IMU device.

Return values:

- **0** – on success.
- **-errno** – Negative errno on failure.

int imu_set_sampling_freq(const struct device *dev, int sampling_rate_hz)

Set the IMU accelerometer and gyroscope sampling frequency.

Parameters:

- **dev** – Pointer to the IMU device.
- **sampling_rate_hz** – Desired sampling rate in Hertz.

Return values:

- **0** – on success.
- **-errno** – Negative errno on failure.

int imu_init(const struct device *dev)

Initialize the IMU device.

Checks device readiness and configures the sampling frequency.

Parameters:

dev – Pointer to the IMU device.

Return values:

- **0** – on success.
- **-ENODEV** – if the device is not ready.

int imu_sensor_value_to_acceleration(const struct imu_data *data, double *acc_out)

calculate the average acceleration from IMU sensor values in m/s^2.

Parameters:

- **data** – Pointer to the IMU sensor data
- **acc_out** – Output for average acceleration in m/s^2. Must be valid pointer to a double.

Return values:

- **0** – on success.
- **-EINVAL** – if `data` or `acc_out` is NULL

int imu_sensor_value_to_orientation(const struct imu_data *data, double dt_s, const double gyro_bias[3], double *orientation)

Calculate the orientation (yaw, pitch, roll) from IMU sensor values.

Uses `CONFIG_IMU_UP_AXIS_*` to remap the body frame so the configured up-axis aligns with world Z, making the result independent of IMU mounting orientation. The two remaining body axes are taken in cyclic order as the local forward (X) and lateral (Y) axes.

Output convention (degrees):

- orientation[0] = yaw (tilt of the forward axis from horizontal)
- orientation[1] = pitch (tilt of the lateral axis from horizontal)
- orientation[2] = roll (rotation about the up axis — the rocket's long axis / flight path)

Yaw and pitch are derived from the accelerometer (gravity-dominated, meaningful only during quasi-static phases). Roll is unobservable from a static accelerometer reading because it is the rotation about the gravity vector itself; it is integrated from the gyroscope.

The caller owns the roll state: `orientation`[2] is read on input as the previous roll angle, advanced by `gyro_up` * dt_s, and written back wrapped to [-180, 180] degrees. Pass `dt_s` <= 0 to leave the roll value untouched (e.g. on the very first sample, before a dt is known).

Parameters:

- **data** – Pointer to the IMU sensor data.
- **dt_s** – Elapsed time in seconds since the previous call, used to integrate roll. Pass 0 to skip integration (yaw and pitch are still updated).
- **gyro_bias** – Optional bias to subtract from the gyro reading before integration, in rad/s. Pass NULL to skip bias correction.
- **orientation** – In/out: [yaw, pitch, roll] in degrees. Yaw and pitch are overwritten; roll is read and updated. Must be a valid pointer to a 3-element double array.

Return values:

- **0** – on success.
- **-EINVAL** – if `data` or `orientation` is NULL.

IMU_NUM_AXES

Number of axes for IMU measurements.

struct imu_data

#include <imu.h>

IMU measurement data structure.

carries the measurement data from the IMU, including accelerometer and gyroscope readings for the x, y, and z axes. This struct is used as a z-bus message payload for IMU data updates

Attitude

Gyro-integrated body-frame gravity tracker. Anchors the body-frame “up” direction from a stationary accelerometer calibration window (typically during `SM_ARMED`), then propagates the

gravity vector through flight using gyro measurements to project body-frame acceleration onto the world vertical axis for the filter.

Mounting orientation is configured via the `CONFIG_IMU_UP_AXIS_*` choice; calibration window length via `CONFIG_IMU_CALIBRATION_SAMPLES`.

int attitude_init(struct attitude *att)

Initialize (or reset) the attitude tracker.

Clears calibration sums, biases, and sets the body-frame gravity vector to the axis selected via `CONFIG_IMU_UP_AXIS_*`, pointing opposite to "up" (i.e. in the direction gravity pulls the rocket).

Parameters:

att – Pointer to tracker state.

Return values:

- **0** – on success.
- **-EINVAL** – if `att` is NULL.

int attitude_calibrate_sample(struct attitude *att, const double accel[3], const double gyro[3])

Add one IMU sample to the calibration accumulator.

Must only be called while the rocket is stationary.

Parameters:

- **att** – Pointer to tracker state.
- **accel** – Body-frame accelerometer reading in m/s^2.
- **gyro** – Body-frame gyroscope reading in rad/s.

Return values:

- **0** – on success.
- **-EINVAL** – if any pointer is NULL.
- **-EALREADY** – if calibration has already been finalized.

int attitude_calibrate_converged(const struct attitude *att)

Query whether the calibration accumulator is ready to finish.

True once the running gyro-bias mean has stopped changing meaningfully between convergence checkpoints, or once a fixed multiple of `CONFIG_IMU_CALIBRATION_SAMPLES` samples have been accumulated (a safety ceiling), whichever comes first. Always false before `CONFIG_IMU_CALIBRATION_SAMPLES` samples have been accumulated.

Parameters:

att – Pointer to tracker state.

Return values:

- **1** – if ready to call attitude_calibrate_finish().
- **0** – if not yet ready.
- **-EINVAL** – if `att` is NULL.

int attitude_calibrate_finish(struct attitude *att)

Finalize calibration and seed the body-frame gravity vector.

Averages the accumulated samples to compute accelerometer bias, gyro bias, and gravity magnitude, then seeds `g_b` from the Kconfig mounting axis. After this call attitude_is_calibrated returns non-zero.

Parameters:

att – Pointer to tracker state.

Return values:

- **0** – on success.
- **-EINVAL** – if `att` is NULL.
- **-ENODATA** – if no samples have been accumulated.

int attitude_update(struct attitude *att, const double accel[3], const double gyro[3], double dt_s, double *accel_vert_out)

Propagate the gravity vector with a new IMU sample and project accelerometer into world vertical.

Subtracts biases, rotates the body-frame gravity vector by the gyro-integrated body rotation (small-angle Rodrigues), renormalizes, and returns the gravity-removed world-frame vertical acceleration (positive = up).

Parameters:

- **att** – Pointer to tracker state.
- **accel** – Body-frame accelerometer reading in m/s^2.
- **gyro** – Body-frame gyroscope reading in rad/s.
- **dt_s** – Elapsed time in seconds since the previous update.
- **accel_vert_out** – Output: world-frame vertical accel in m/s^2, gravity-removed (positive = up).

Return values:

- **0** – on success.
- **-EINVAL** – if any pointer is NULL or `dt_s` <= 0.
- **-ENODATA** – if calibration has not been finalized.

int attitude_is_calibrated(const struct attitude *att)

Query whether calibration has been finalized.

Parameters:

att – Pointer to tracker state.

Return values:

1 – if calibrated, 0 if not, -EINVAL if `att` is NULL.

ATTITUDE_NUM_AXES

Number of axes handled by the attitude tracker.

struct attitude

#include <attitude.h>

Attitude tracker state.

Barometer

Interface for the barometric pressure sensor. Can compute altitude estimates from pressure readings.

ZBUS_CHAN_DECLARE(baro_data_chan)

ZBUS channel for baro data.

int baro_measure(const struct device *dev)

Measure temperature and pressure from the barometric sensor and publish the data to the z-bus.

Parameters:

dev – Pointer to the barometric sensor device.

Return values:

- **0** – on success.
- **-EINVAL** – if `dev` is NULL
- **-errno** – Other negative errno on failure.

int baro_init(const struct device *dev)

Initialize the barometric pressure sensor.

Checks device readiness and configures the oversampling rate.

Parameters:

dev – Pointer to the barometric sensor device.

Return values:

- **0** – on success.
- **-ETIMEDOUT** – if the device is not ready.
- **-EIO** – if oversampling configuration fails.

int baro_set_reference(double ref_kpa)

Set the ground-level reference pressure.

Must be called before baro_sensor_value_to_altitude to establish the zero-altitude baseline. Typically called once at startup with the first valid pressure reading.

Parameters:

ref_kpa – Ground-level pressure in kilopascals.

Return values:

- **0** – on success.
- **-EINVAL** – if `ref_kpa` is not positive.

int baro_sensor_value_to_altitude(const struct sensor_value *press, double *altitude_out)

Convert a pressure reading to altitude AGL.

Uses the hypsometric formula (ISA troposphere model) with the reference pressure set by baro_set_reference.

Parameters:

- **press** – Barometric pressure as sensor_value.
- **altitude_out** – Altitude in meters above the reference level.

Return values:

- **0** – on success.
- **-EINVAL** – if `press` or `altitude_out` is NULL.

struct baro_data

#include <baro.h>
baro measurement data structure.

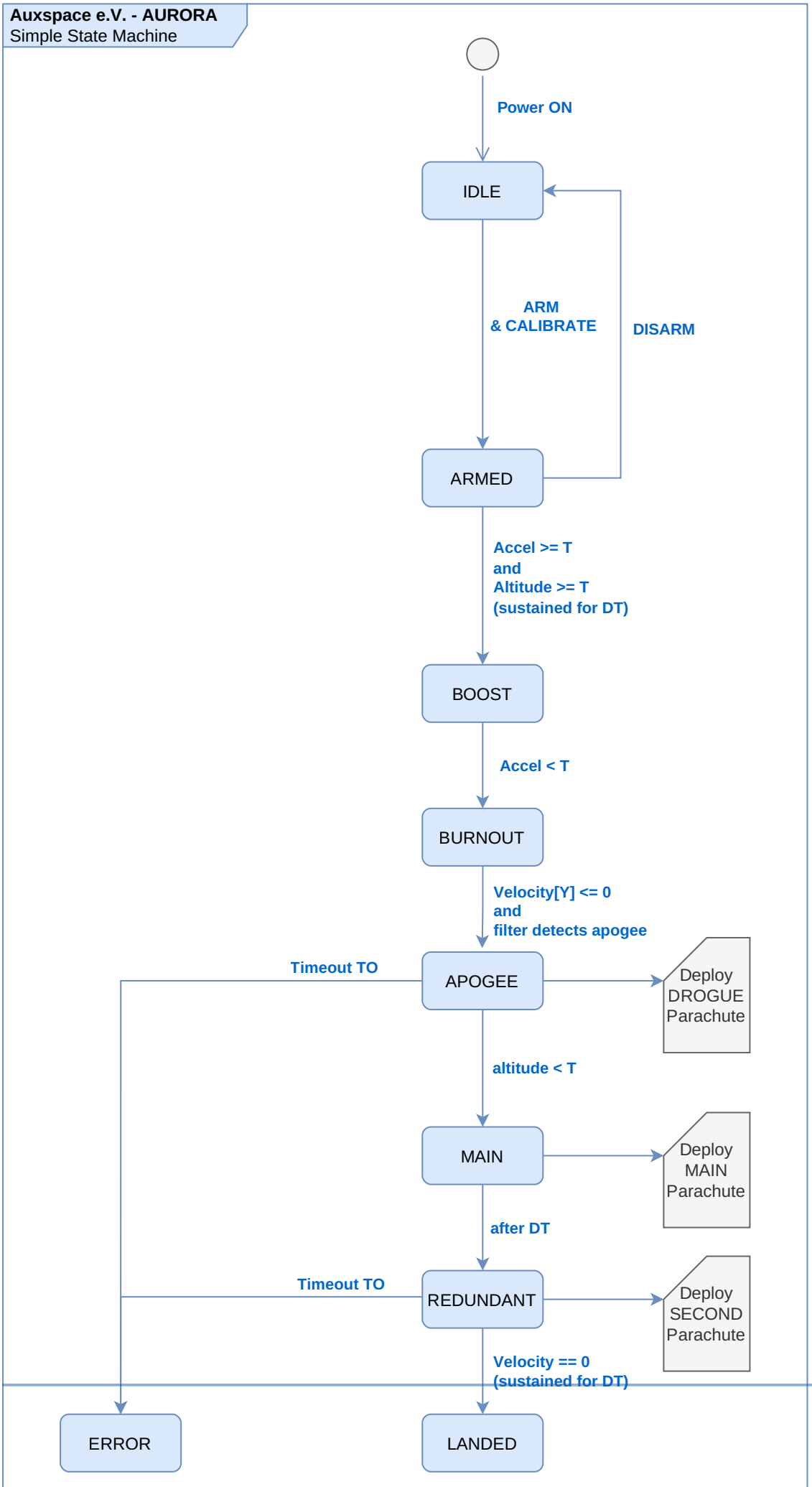
carries the measurement data from the baro including temperature and pressure readings. This struct is used as a z-bus message payload for baro data updates

State Machine

The state machine library provides a generic interface for initializing, updating, and querying the flight state. As almost everything else in AURORA, it features a dynamic selection of state machine types via Kconfig `CONFIG_AURORA_STATE_MACHINE_TYPE`. Currently only the simple state machine is implemented and it uses the following flight sequence:

Simple State Machine

The simple state machine implementation defines a 9-state flight sequence driven by sensor thresholds.


Auxspace e.V. - AURORA
Simple State Machine
Power ON
IDLE
ARM
& CALIBRATE
DISARM
ARMED
Accel >= T
and
Altitude >= T
(sustained for DT)
BOOST
Accel < T
BURNOUT
Velocity[Y] <= 0
and
filter detects apogee
Timeout TO
APOGEE
Deploy
DROGUE
Parachute
altitude < T
MAIN
Deploy
MAIN
Parachute
after DT
Timeout TO
REDUNDANT
Deploy
SECOND
Parachute
Velocity == 0
(sustained for DT)
ERROR
LANDED

Signals

Signal	Comment
ARM	*ARM* signal from extern. Arms the pyro channels as well
DISARM	*DISARM* signal from extern. Disarms the pyro channels as well

Sensor Readings

Sensor Reading	Comment
T_{AB}	Acceleration needed to go from ARMED to BOOST
T_{H}	Altitude needed to go from ARMED to BOOST
T_{BB}	Acceleration needed to go from BOOST to BURNOUT
T_{M}	Altitude needed to go from APOGEE to MAIN
T_{L}	Velocity needed to signal LANDED

Timers

Timer	Comment
DT_{AB}	Time that T_AB and T_H shall be asserted for
DT_{L}	Time that T_L shall be asserted

Timeouts

Timeout	Comment
TO_A	Timeout for APOGEE state
TO_R	Timeout for REDUNDANT state

State transitions are also driven by sensor thresholds configured via Kconfig (boost acceleration, main descent height, apogee timeout, etc.).

Shell Commands

Enabling `CONFIG_AURORA_STATE_MACHINE_SHELL` registers the `state_machine` command group. Audit-log commands are only available when `CONFIG_AURORA_STATE_MACHINE_AUDIT` is also enabled.

Command	Description
`state_machine status`	Print the active state-machine implementation and its current state.
`state_machine transition <STATE>`	Force a transition. The state name completes via tab. Because the state machine exposes no arbitrary setter, this deinitializes and reinitializes the machine, landing it in `IDLE`; a warning is printed when the requested target is not `IDLE`. Ground testing only.
`state_machine audit`	Dump the audit log (timestamped transitions and events). Requires `CONFIG_AURORA_STATE_MACHINE_AUDIT`.
`state_machine audit_clear`	Clear the audit log. Requires `CONFIG_AURORA_STATE_MACHINE_AUDIT`.

Valid state names for `transition` are `IDLE`, `ARMED`, `BOOST`, `BURNOUT`, `APOGEE`, `MAIN`, `REDUNDANT`, `LANDED` and `ERROR`.

⚠ Warning

`state_machine transition` bypasses normal flight logic and resets the machine. Do not use in flight.

API Reference

enum sm_type

Identifier of the active state machine implementation.

Each implementation defines its own `sm_state` enum, so any external consumer (ground station, post-flight tooling) needs to know which implementation produced a given state value before it can decode it.

Values:

enumerator SM_TYPE_SIMPLE

Simple backend (lib/state/simple.c).

enum sm_error_reason

Reasons the state machine can enter `SM_ERROR`.

Passed to the error callback so the application can decide per cause whether to recover (return 0 → back to IDLE) or hold the error state (non-zero, e.g. a pre-flight interlock the operator must acknowledge by disarming) and how to signal the operator (LED, buzzer, ...).

Values:

enumerator SM_ERR_UNKNOWN

Unspecified error.

enumerator SM_ERR_LOG_OFFLINE

Flight recorder unavailable while arming or armed.

enumerator SM_ERR_APOGEE_TIMEOUT

No descent detected within TO_A.

enumerator SM_ERR_REDUNDANT_TIMEOUT

No landing detected within TO_R.

typedef int (*sm_error_cb_t)(enum sm_error_reason reason, void *args)

Callback invoked when the state machine encounters an error.

The implementation can define specific recovery logic.

Param reason:

Why the state machine entered `SM_ERROR`.

Param args:

Pointer to an implementation-specific config structure.

Return:

0 if the error was mitigated (state machine returns to IDLE), negative errno to hold `SM_ERROR` (the callback is re-invoked on every update until it mitigates or the system is disarmed).

const char *sm_state_str(enum sm_state state)

Return a human-readable name for the given state.

Parameters:

state – State value.

Returns:

Pointer to a static string, or “UNKNOWN” if invalid.

const char *sm_error_reason_str(enum sm_error_reason reason)

Return a human-readable name for the given error reason.

Parameters:

reason – Error reason value.

Returns:

Pointer to a static string, or “UNKNOWN” if invalid.

void sm_init(const struct sm_thresholds *cfg, struct sm_error_handling_args *err_hdl)

Initialize the rocket state machine.

This function prepares the state machine, loads the threshold configuration, initializes internal timers, and sets the initial state to `SM_IDLE`.

Parameters:

- **cfg** – Pointer to a threshold configuration structure.
- **err_hdl** – Pointer to an error handling configuration (callback + args), or NULL.

void sm_deinit(void)

Deinitialize the rocket state machine.

This function resets the state machine, unloads the threshold configuration, stops internal timers, and sets the initial state.

void sm_update(const struct sm_inputs *inputs)

Update the state machine using current sensor readings.

Function evaluates sensor data and executes state transitions according to the flight logic diagram. Must be called regularly (e.g. at sensor update rate).

Parameters:

inputs – Pointer to populated sensor readings.

enum sm_state sm_get_state(void)

Retrieve the current state of the state machine.

Returns:

Current state (usually an enum implementation in state implementation).

enum sm_type sm_get_type(void)

Identify which state machine implementation is active.

External consumers use it to pick the right `sm_state` enum mapping.

Returns:

Active state machine type ID.

void sm_get_inputs(struct sm_inputs *out)

Retrieve the most recent inputs the state machine evaluated.

When CONFIG_FILTER is enabled, `altitude` and `velocity` are the Kalman-filtered values; the remaining fields are passed through from the last sm_update() call. When CONFIG_FILTER is disabled, all fields are the raw sm_update() inputs (and `velocity` is whatever the caller set, typically 0).

Before the first sm_update() call the returned struct is zeroed.

Parameters:

out – Destination struct, must be non-NULL.

void sm_update_force(enum sm_state transition_to)

Update the state machine with force. No further checks are done.

Parameters:

transition_to – State to transition.

AURORA_STATE_BACKEND_INTERNAL

struct sm_thresholds

#include <simple.h>

Threshold configuration for the rocket state machine.

Defines thresholds for state transitions based on orientation, altitude, acceleration, and timing.

struct sm_inputs

#include <simple.h>

Sensor input structure for the state machine.

These values must be filled each update cycle to evaluate state transitions.

struct sm_error_handling_args

#include <state.h>

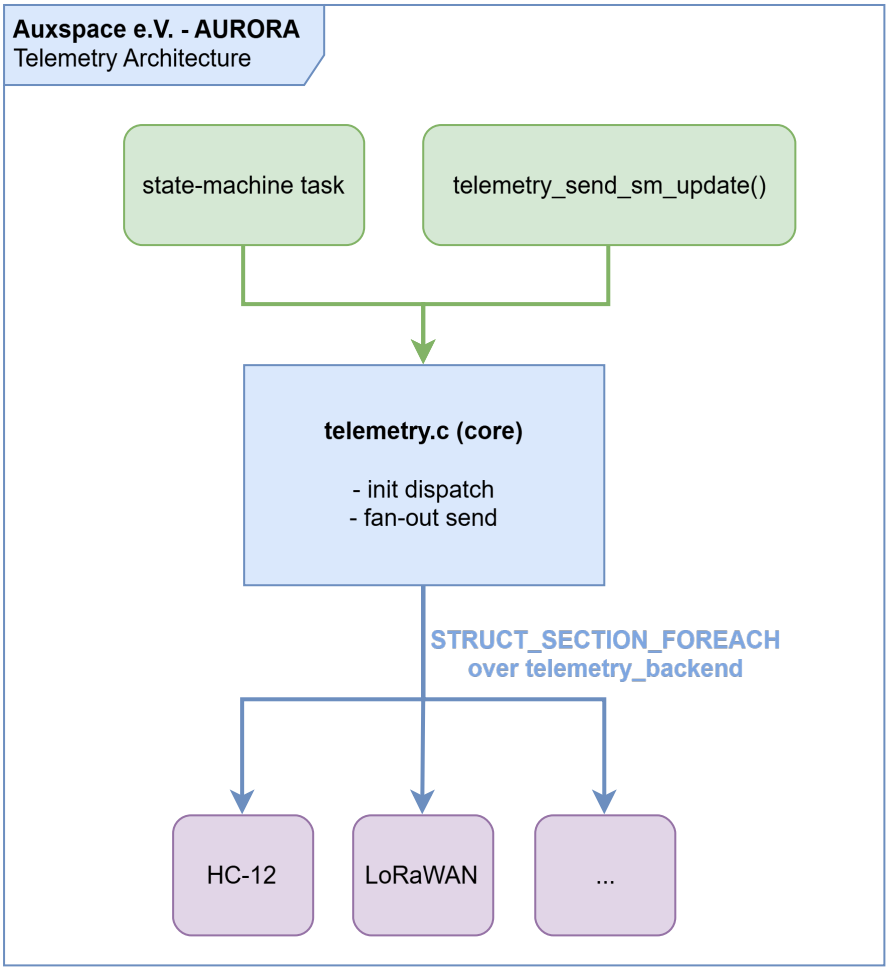
Error handling configuration for the state machine.

Telemetry

The telemetry library is the downlink path for in-flight data. It is a small dispatcher: every backend that registers a `telemetry_backend` at link time receives each outgoing message. Backends own their own framing, transport, worker threads, and any backend-specific rate limiting.

Today only the HC-12 433 MHz UART-RF bridge backend ships in-tree, but the API is transport-agnostic: a LoRaWAN, CAN-tunnel, or any other backend can be added without touching the dispatcher or callers.

Architecture



The dispatcher is synchronous and runs in the caller's thread: it walks the iterable section and invokes each backend's `send_sm_update` hook inline. Backends that need to do real work (UART writes, radio TX) must offload to their own worker thread and return immediately. The dispatcher itself never blocks.

Backends

- **HC-12** (`CONFIG_AURORA_TELEMETRY_HC12`): transparent UART ↔ 433 MHz RF bridge. Selects its UART via the chosen `auxspace,telemetry-uart` node. The module itself is provisioned out of band on the bench (channel, air baud, TX power); firmware only opens the UART. See [HC-12 wire frame] and [HC-12 threading and rate limiting].

Adding a new backend

A backend is two things: a vtable filled with the operations it supports, and a single `TELEMETRY_BACKEND_DEFINE` invocation to register it at link time.

```
#include <aurora/lib/telemetry.h>

static int my_init(void) { /* ... */ return 0; }

static int my_send_sm_update(enum sm_state state, enum sm_type type,
                             const struct sm_inputs *inputs)
{
    /* Frame and enqueue. Must not block. Return:
     *   0       on accept,
     *   -EAGAIN if throttled,
     *   -ENOMEM if your TX queue is full,
     *   -ENODEV if the transport is not ready.
     */
    return 0;
}

static const struct telemetry_backend_api my_api = {
    .init           = my_init,
    .send_sm_update = my_send_sm_update,
};

TELEMETRY_BACKEND_DEFINE(my_backend, &my_api);
```

Add a `CONFIG_AURORA_TELEMETRY_<BACKEND>` symbol under `lib/telemetry/Kconfig` and a conditional `zephyr_library_sources(...)` block in `lib/telemetry/CMakeLists.txt` to compile it in. No changes to the dispatcher or to callers (`telemetry_send_sm_update`) are needed.

HC-12 wire frame

The HC-12 backend frames each state-machine update as a small self-describing packet so the ground station can resync after RF corruption. All multi-byte fields are little-endian.

Offset	Size	Field
0	1	magic0 = `0xA5`
1	1	magic1 = `0x5A`

Offset	Size	Field
2	1	type (see below)
3	1	len (payload bytes that follow, excluding CRC)
4	len	payload
4 + len	2	CRC-16/CCITT (init `0xFFFF`) over bytes `[2 .. 4+len-1]`

Packet types:

Value	Name	Payload
`0x01`	`SM_UPDATE`	State-machine snapshot (see below)

`SM_UPDATE` payload (36 bytes):

Offset	Size	Type	Field
0	4	`u32`	`timestamp_ms` (system uptime, low 32 bits)
4	1	`u8`	`state` (`sm_state`)
5	1	`u8`	`armed` (0/1)
6	1	`u8`	`sm_type` (`sm_type` identifies the `state` enum mapping the firmware is using)
7	1	`u8`	reserved (zero)
8	4	`f32`	`altitude` (m)

Offset	Size	Type	Field
12	4	`f32`	`acceleration` (m/s², total)
16	4	`f32`	`accel_vert` (m/s², body-vertical)
20	4	`f32`	`velocity` (m/s)
24	12	`f32[3]`	`orientation` (roll/pitch/yaw, rad)

At 10 Hz the link runs at roughly 420 B/s, about 44 % of a 9600-baud HC-12 air link, leaving headroom for re-tries and other packet types.

HC-12 threading and rate limiting

The HC-12 backend hands every outgoing frame to a dedicated worker thread via a bounded FIFO message queue. The producer (`telemetry_send_sm_update()`, called from the state-machine task) never blocks:

- Frames are framed and CRC'd inline on the caller's stack, then posted with `K_NO_WAIT`. A full queue returns `-ENOMEM` and drops the frame rather than stalling the SM thread.
- The worker drains the queue and writes bytes with `uart_poll_out`. Keeping it on a low-priority thread (default priority 10) ensures telemetry can never preempt flight-critical threads (sensors and the state machine run at priority 5–6).
- Optional rate limiting drops frames before they touch the queue or the UART. Useful when the SM tick rate is higher than the air link can carry comfortably (the default 0 disables it).

Tunables (under `AURORA_TELEMETRY_HC12`):

Kconfig	Default	Purpose
`AURORA_TELEMETRY_HC12_QUEUE_DEPTH`	16	Maximum queued frames before overflow drops.
`AURORA_TELEMETRY_HC12_MIN_INTERVAL_MS`	0	Minimum spacing between accepted SM

Kconfig	Default	Purpose
		updates (ms). 0 = unlimited.
`AURORA_TELEMETRY_HC12_STACK_SIZE`	1024	Worker thread stack size (bytes).
`AURORA_TELEMETRY_HC12_THREAD_PRIORITY`	10	Worker thread priority. Keep numerically above flight threads (priority 5) so telemetry never preempts them.

Device-tree

The HC-12 backend is bound to a devicetree node with the compatible `auxspaceev,hc12` (see `dts/bindings/hc12/auxspaceev,hc12.yaml`). The node carries a phandle to the host UART and, optionally, the `SET` line:

Minimal node. Transparent downlink only, no runtime AT support:

```
/ {
   hc12: hc12 {
      compatible = "auxspaceev,hc12";
      uart = <&uart1>;
      status = "okay";
   };
};
```

Add `set-gpios` to also enable runtime provisioning through the shell (see [Provisioning shell]):

```
hc12: hc12 {
    compatible = "auxspaceev,hc12";
    uart = <&uart1>;
    set-gpios = <&gpio0 6 GPIO_ACTIVE_LOW>;
    status = "okay";
};
```

Boards that do not mount an HC-12 leave the node disabled (or omit it entirely); the `AURORA_TELEMETRY_HC12` symbol is then unselectable (`DT_HAS_AUXSPACEEV_HC12_ENABLED` resolves

to `n`) and the backend is compiled out. The default `sensor_board_v2` DTS ships the node `status = "disabled"` so each application opts in explicitly via an overlay.

Provisioning

The HC-12 stores channel, air baud, TX power and FU mode in its own non-volatile flash. **Settings persist across power cycles**, so the firmware never sends AT commands at boot: it only configures `SET` (when wired) to transparent mode and opens the UART. Provision the module *once*, when something needs to change.

The HC-12 enters AT-command mode while its `SET` pin is pulled low. In AT mode the module always speaks 9600 baud regardless of the transparent-mode air baud. There are two ways to provision it:

1. **On the bench** with a USB-serial adapter: works on any board, no firmware support required, no risk of stalling the avionics bus.
2. **From the firmware shell**: works on boards whose HC-12 has its `SET` line wired to a GPIO and that build with `CONFIG_AURORA_TELEMETRY_HC12_SHELL=y`.

Bench provisioning

Pull `SET` low, connect a USB-serial adapter at 9600 baud, and send:

Command	Effect
`AT`	Sanity check; module replies `OK`.
`AT+B9600`	Set air baud (must match host `current-speed`).
`AT+Cnnn`	Set channel (`001`-`127`). Match the ground module.
`AT+P8`	Maximum TX power (~20 dBm).
`AT+FU3`	Mode 3 (good range/throughput compromise; default).
`AT+RX`	Read back the current settings.

Command	Effect
`AT+DEFAULT`	Restore factory defaults.

Release `SET` to exit AT mode. The ground-side HC-12 must use the same channel, baud, and FU mode.

Provisioning shell

Enable `CONFIG_AURORA_TELEMETRY_HC12_SHELL=y` (which selects `CONFIG_AURORA_TELEMETRY_HC12_AT=y`) and the `telemetry hc12` command tree appears on the Zephyr shell:

Command	Effect
`telemetry hc12 info`	Issue `AT+RX` and print channel/baud/power/mode.
`telemetry hc12 channel <1..127>`	`AT+C<nnn>`.
`telemetry hc12 baud <rate>`	`AT+B<rate>` for `1200|2400|4800|9600|19200|38400|57600|115200`. Warns that `current-speed` in DT must be updated and the board rebooted to keep the host UART in sync.
`telemetry hc12 power <1..8>`	`AT+P<n>`; `8` is +20 dBm (maximum).
`telemetry hc12 mode <1..4>`	`AT+FU<n>`.
`telemetry hc12 reset`	`AT+DEFAULT`.
`telemetry hc12 at <command>`	Raw escape hatch, e.g. `telemetry hc12 at AT+V`.

The shell handler:

- **Refuses every command unless the state machine is in IDLE.** Reprovisioning while armed would silence the downlink mid-flight and stall the UART worker for ~200 ms; that is never the right thing to do in any other state.
- **Holds a mutex on the HC-12 UART** for the duration of each AT exchange. The TX worker thread blocks on the same mutex per outgoing frame, so transparent-mode bytes can never collide with an AT command. Frames generated while AT is in progress queue up normally and drain when the lock is released.
- **Restores the host UART baud on every exit path** (including errors), so a failed `AT+B` cannot leave the host out of sync with the radio.

If the `set-gpios` property is absent from the HC-12 node, every command returns `-ENOTSUP` and prints a hint pointing at the bench flow.

Usage from the application

`main.c` calls `telemetry_init()` once at boot, then `telemetry_send_sm_update()` once per state-machine tick:

```
#if defined(CONFIG_AURORA_TELEMETRY)
    (void)telemetry_init();
#endif

/* ... inside the state-machine loop, after sm_update(): */
#if defined(CONFIG_AURORA_TELEMETRY)
    struct sm_inputs sm_in;
    sm_get_inputs(&sm_in);
    (void)telemetry_send_sm_update(state, &sm_in);
#endif
```

The return value of `telemetry_send_sm_update()` is the first non-zero error any backend returned; remaining backends are still called. Callers in the hot path typically ignore it.

API Reference

int telemetry_init(void)

Initialise all registered telemetry backends.

Return values:

0 – on success, or the first non-zero return from a backend.

int telemetry_send_sm_update(enum sm_state state, enum sm_type type, const struct sm_inputs *inputs)

Fan out a state-machine update to all registered backends.

Safe to call from any thread context. Never blocks — backends must enforce that themselves (typically by dropping on overflow).

Parameters:

- **state** – Current flight state.
- **type** – Active state machine implementation ID (see sm_get_type). Forwarded so the receiver can decode `state` without prior agreement.
- **inputs** – Current SM inputs snapshot (see sm_get_inputs).

Return values:

- **0** – if every backend accepted the message.
- **<0** – the first error returned by any backend (others still tried).

TELEMETRY_BACKEND_DEFINE(_name, _api)

Register a telemetry backend (link-time).

Parameters:

- **_name** – Unique C identifier for this backend.
- **_api** – Pointer to a telemetry_backend_api vtable.

struct telemetry_backend_api

#include <telemetry.h>

Backend operations vtable.

All members are optional; NULL slots are skipped by the dispatcher.

struct telemetry_backend

#include <telemetry.h>

Backend descriptor. Collected at link time.

Tools

The `aurora/tools` directory collects standalone Python scripts used during development, testing and log analysis. They are not part of the firmware build and are invoked directly with `python3`.

All scripts share the same copyright header and SPDX identifier as the rest of the repository and target Python 3.10+.

> **Note**
>
> Pip requirements are installed via `pip install -r tools/requirements.txt`

gen_flight_replay.py

Source: `tools/gen_flight_replay.py`

Convert a recorded AURORA flight CSV into a generated C source file that embeds the samples as constant arrays. The generated file is compiled into the firmware and consumed by the **replay backend** of `fake_sensors`, so that simulated runs (e.g. on `native_sim` or with the `sim` snippet) can be driven by *real* flight data instead of the analytic profile.

Background

When `CONFIG_AURORA_FAKE_SENSORS=y` is set, the IMU and baro polling threads are replaced by a synthetic data source. There are two backends:

Backend	Kconfig	What it publishes
Analytic profile	`AURORA_FAKE_SENSORS_SYNTH`	An ISA-troposphere ascent/descent curve generated on the fly.
Replay	`AURORA_FAKE_SENSORS_REPLAY`	Samples from a previously recorded flight, embedded at build time.

The replay backend (`fake_sensors_replay.c`) expects three arrays:

```
struct replay_imu_sample  { uint64_t t_ns; float x, y, z; };
struct replay_baro_sample { uint64_t t_ns; float pres_kpa, temp_c; };

const struct replay_imu_sample  replay_accel[];
const struct replay_imu_sample  replay_gyro[];
const struct replay_baro_sample replay_baro[];
```

`gen_flight_replay.py` is the small code-generator that produces the C file holding these arrays. It is invoked automatically from the sensor_board `CMakeLists.txt` whenever the replay backend is enabled — you normally do not call it by hand.

Inputs

The script reads the **`flights.csv`** file produced by the data logger's CSV converter. Each row is expected to carry a `timestamp_ns` column plus optional `accel_{x,y,z}`, `gyro_{x,y,z}` and `baro_pres` / `baro_temp` columns. Empty cells are simply skipped, which is convenient because the logger only fills the columns of the sensor that produced a given row.

Optionally, a **`state_audit`** file from the same flight may be passed in. This is the human-readable transition log written by the flight state machine. The script parses the `BOOST` and `LANDED` transitions out of it and trims the sample arrays to a window of:

```
[BOOST  - 4 s, LANDED + 4 s]
```

Trimming keeps the embedded array small (and therefore the firmware image small) while still leaving enough margin on either side for attitude calibration and post-landing checks.

> **Note**
>
> Despite the `Time (ms)` header in the audit file, the recorded values are actually in nanoseconds and match `timestamp_ns` in the CSV one-to-one.

Output

A single C source file with:

- A header comment listing the source CSV, the sample counts and the total time span (and, if applicable, the trim window).
- Three array definitions: `replay_accel`, `replay_gyro`, `replay_baro`.
- A `<name>_len` constant for each array (computed via `ARRAY_SIZE`).

All timestamps in the generated file are **rebased to start at zero** — the smallest timestamp across all three streams is subtracted from every sample. This way the replay engine can work in elapsed time without caring about the absolute uptime of the original flight.

Usage

```
python3 tools/gen_flight_replay.py \
    --input        flights.csv \
    --output       replay_data.c \
    [--state-audit state_audit]
```

Argument	Description
`--input`	Path to a `flights.csv` produced by the data logger.
`--output`	Destination path for the generated C source file.
`--state-audit`	Optional. State machine audit log from the *same* flight. When present, samples are trimmed to `[BOOST - 4 s, LANDED + 4 s]`.

The script exits with a non-zero status (and a message on `stderr`) if:

- the CSV is missing accelerometer, gyroscope or barometer samples, or
- a `--state-audit` is supplied but contains no `BOOST` / `LANDED` transition, or
- the trim window ends up empty (typically a sign that the audit and CSV timestamps come from different runs).

Build integration

You usually do not run `gen_flight_replay.py` directly. The sensor_board `CMakeLists.txt` adds a custom command that re-runs it whenever the input CSV (or, if present, the sibling `state_audit` file) changes:

```
add_custom_command(
    OUTPUT  ${REPLAY_OUT}
    COMMAND ${PYTHON_EXECUTABLE} ${REPLAY_GEN}
        --input ${REPLAY_INPUT}
        --output ${REPLAY_OUT}
```

```
        ${REPLAY_AUDIT_ARGS}
    DEPENDS ${REPLAY_INPUT} ${REPLAY_GEN} ${REPLAY_AUDIT_DEPS}
    COMMENT "Generating replay_data.c from ${REPLAY_INPUT_REL}"
)
```

To swap recordings, change the Kconfig string `CONFIG_AURORA_FAKE_SENSORS_REPLAY_INPUT` (path is relative to the aurora module root). If a `state_audit` file sits next to the chosen CSV, it is picked up automatically.

Example

Generate a replay source file from a recorded multimeter flight and trim it to the powered-flight window:

```
python3 ./tools/gen_flight_replay.py \
    --input        flight_logs/multimeter/2026-05-03/flight1/
flights.csv \
    --output       /tmp/replay_data.c \
    --state-audit  flight_logs/multimeter/2026-05-03/flight1/
state_audit
```

To then build a `native_sim` image that replays this flight:

```
west build -p -b native_sim -S sim aurora/sensor_board \
    -- -DCONFIG_AURORA_FAKE_SENSORS_REPLAY=y
./build/zephyr/zephyr.exe
# in the Zephyr shell:
uart:~$ sim launch
```

Requirements

- Python 3.10+
- No third-party packages (uses `argparse`, `csv` and `re` only).

pad_link_central_example.py

Source: `tools/pad_link_central_example.py`

Minimal BLE central for the AURORA pad-link.

This script is the smallest thing that talks to a rocket running `CONFIG_AURORA_PAD_LINK=y`. It scans for the rocket's advertised service, connects, reads the static characteristics (board name, SM type, and board capabilities), prints what the board supports, and then subscribes to / polls only the characteristics that are actually present according to the capability register.

It is meant as a *reference*, not a finished ground station. Use it to:

- test the rocket-side firmware after enabling the pad link (you should see at least the board id, capabilities, and the IDLE state),
- see how to decode the wire layouts documented in Pad Link in a real client,
- copy/paste pieces into a fuller UI (a Tk dashboard, a logger, the launchrail's own central).

⚠ Warning

On ESP32-S3 boards, running this script in `notify`-mode against the rocket can make the flight computer **freeze** after a few seconds to a few minutes of a live connection (you will stop seeing new values, and its USB console goes dead until you power-cycle it). This is a known, unresolved firmware bug, not a problem with your setup; see the warning in Pad Link. Don't rely on this link during a countdown or flight.

Wire contract

Mirrors aurora/lib/pad_link/pad_link.c.

❗ Important

If the C-side struct or a UUID changes, the constants here must change too.

Characteristics:

Characteristic	Description
SERVICE_UUID	primary service to filter on during scan
UUID_BOARD	UTF-8 string, board identifier

Characteristic	Description
UUID_SMTYPE	uint8, identifies which sm_state enum is in use
UUID_BOARDCAP	uint32 LE, capability register (see below)
UUID_STATE	uint8, current SM state (notify-capable)
UUID_COMP	packed struct, computed kinematics (notify-capable)
UUID_BARO	packed struct, baro pressure + temperature (notify-capable)
UUID_IMU6	packed struct, 6-DoF accel + gyro (notify-capable)
UUID_INNER_TEMP	packed struct, inner temperature (notify-capable)
UUID_RAW	packed struct, raw IMU+baro — deprecated, not used here

Board capabilities

After connecting the script reads `UUID_BOARDCAP` (characteristic `a0`), a `uint32` little-endian register that describes which sensor characteristics carry valid data on this specific board:

Bits	Field	Values
[2:0]	IMU type	0=none, 1=6-DoF, 2=9-DoF
[3]	Accel	1 = accelerometer valid (a2)
[4]	Gyro	1 = gyrometer valid (a3)
[8]	Baro	1 = barometer valid (a1)

Bits	Field	Values
[9]	Inner temp	1 = inner temperature valid (a7)
[10]	Motor temp	1 = motor temperature valid (a8, planned)
[11]	Hull temp	1 = hull temperature valid (a9, planned)
[16]	GPS/GNSS	1 = GPS valid (a6, planned)

The script uses this register to decide which optional characteristics to subscribe to or poll, so it works correctly on both hardware revisions (µMETER v1 with LPS22HH, µMETER v2 with BMP581 + extra sensors).

Payload layouts (all little-endian, micro-units ÷ 1 000 000)

```
comp:       struct.unpack("<Ifffffff", data[:28])  → uptime_ms, alt_m,
vel_m_s, yaw, pitch, roll, az
baro:       struct.unpack("<Iqq",     data[:20])  → uptime_ms,
temp_µ°C, press_µPa
imu6:       struct.unpack("<Iqqqqqq", data[:52])  → uptime_ms,
ax,ay,az µm/s², gx,gy,gz µrad/s
inner_temp: struct.unpack("<Iq",      data[:12])  → uptime_ms,
temp_µ°C
boardcap:   struct.unpack("<I",       data[:4])   → flags
```

Prerequisites

- Python 3.9+
- `pip install bleak`
- A BLE-capable host (built-in adapter on Linux/macOS/Windows, or a plugged-in USB BLE dongle).
- On Linux the user usually needs to be in the `bluetooth` group, or the script has to be run with `sudo`.

Running

```
python3 pad_link_central_example.py
```

You should see the board name, a capability summary, and then a stream of state and telemetry prints. The script exits after 60 seconds; pass `--duration` to change this.

Notes for first-time users

- Default ATT MTU (23 B) is enough: `read_gatt_char` and notifications transparently do *long reads* for payloads larger than the MTU. Negotiating a bigger MTU (`BleakClient(..., mtu=247)`) just makes them faster.
- The first read of `UUID_STATE` before the rocket's state machine has run even once returns 0 (= IDLE). Values stabilise within milliseconds of boot.
- Disconnects are normal and silent. The rocket re-starts advertising automatically. Wrap the body in a reconnect loop if you want the script to survive them.

plot_flight_data.py

Source: `tools/plot_flight_data.py`

Plot AURORA flight data — recorded telemetry from the flight computer or simulated streams produced by `sim_flight_kalman.py`.

What it does

This is the dedicated plotting tool for the AURORA flight stack. It covers two use cases:

- **Standalone CLI:** `python3 plot_flight_data.py --flight DIR` segments the log under `DIR` into individual flights on `ARMED → BOOST` transitions and produces one multi-panel plot per flight, with state-machine transitions drawn as labelled vertical markers on every panel. `DIR` is expected to contain `flights.influx` (line-protocol telemetry) and `state_audit` (state-machine transition log).
- **Imported module:** `sim_flight_kalman.py` imports the module and calls `plot_flight(...)` to render the seven-panel plot of its simulated output.

The tool does not run the Kalman filter or attitude tracker itself. What it does carry, on top of pure plotting, is the log-loading and replay glue needed to turn raw telemetry into the curves that the plot panels expect — pressure → altitude conversion, NIS-gate replay against the logged Kalman

altitude, the three-vote apogee replay, and the boost/landed-recovery heuristics for logs whose state machine never reached `BOOST`.

> **Note**
>
> The replay panels (NIS gate, apogee votes) approximate the firmware behaviour from logged streams alone. The on-board covariance is not telemetered, so the gate is evaluated under a steady-state assumption. Treat the markers as a sanity check, not as a perfect re-run.

Usage

```
python3 tools/plot_flight_data.py --flight DIR
                                  [--theme {light,dark,both}]
                                  [--show] [--disable-votes]
                                  [--pre-boost SECONDS]
                                  [--post-end SECONDS]
                                  [--title TITLE]
                                  [--trim [SECONDS]]
                                  [--r-meas R_MEAS]
```

Data selection

- `--flight DIR` — required; flight directory containing `flights.influx` and `state_audit`.
- `--pre-boost SECONDS` — seconds of pre-boost data to keep per flight (default `10`).
- `--post-end SECONDS` — seconds of data to keep after the flight close-out transition, trimming the long post-parachute tail (default `2`).

Replay

- `--r-meas R_MEAS` — assumed Kalman measurement variance (m²) used to approximate the NIS gate from logged baro vs `sm_pose`. Match `CONFIG_FILTER_R_MILLISCALE / 1000` for the flight build (default `6.0`).
- `--disable-votes` — drop the apogee-vote panel (useful when the logged streams are known to be unreliable, e.g. a stuck pointer in the logger).

Output

- `--theme {light,dark,both}` — render plots matching the Furo Sphinx docs theme (default `both`).
- `--show` — open the plots in an interactive matplotlib window in addition to writing them to disk.
- `--title TITLE` — override the plot title (applied to every plot the run produces).
- `--trim [SECONDS]` — write a trimmed copy of `flights.influx` next to the original, covering each flight's plot window expanded by `SECONDS` on each side (default `10` when `--trim` is given without a value). Useful for shrinking long pad-time logs before sharing them.

Plots are written to `flight<N>.png` in the current directory, with `_light` / `_dark` suffixes when `--theme both` is used.

Reuse from other scripts

The plotting and log-handling helpers are designed to be importable. The most useful entry points are:

- `parse_influx(path)`, `parse_state_audit(path)`, `segment_flights(events)` — load telemetry and split a log into flights.
- `slice_real_flight(...)` — trim raw streams to one flight window.
- `compute_log_votes(sliced)`, `compute_log_gated(sliced, r_meas)` — replay the apogee votes and NIS gate from logged streams.
- `altitude_to_pressure(h)` / `pressure_to_altitude(p, p_ref)` — ISA conversions.
- `plot_flight(...)` — seven-panel plot used by `sim_flight_kalman.py`.
- `plot_raw_flight(...)` — multi-panel raw-telemetry plot used by the CLI above.

Requirements

- Python 3.10+
- `numpy`, `matplotlib`

rec_zephyr.py

Source: `tools/rec_zephyr.py`

MicroPython-based ground station for receiving AURORA telemetry over an HC-12 radio link during bench tests.

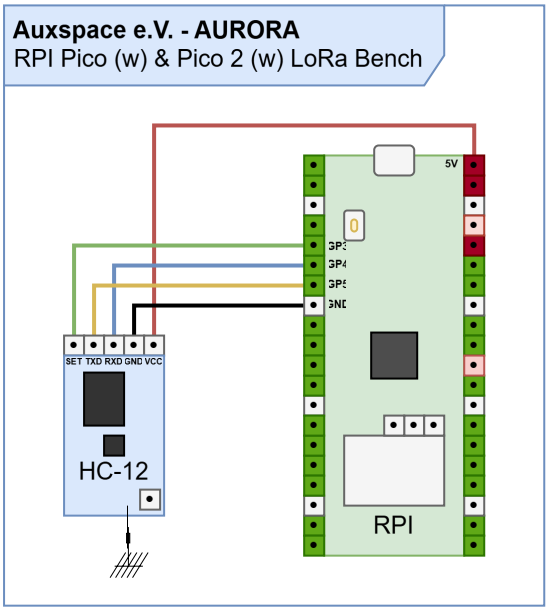
Background

The flight computer broadcasts state-machine updates over an HC-12 radio using the wire format defined in `aurora/lib/telemetry/hc12/hc12.c`. `rec_zephyr.py` runs on a Raspberry Pi Pico flashed with MicroPython, listens to a paired HC-12 over UART, validates each frame and prints a human-readable line per packet to the REPL.

It is intentionally self-contained: no host PC, no extra dependencies, no logging stack. The goal is to confirm that frames are being received and decoded correctly before integrating a more sophisticated tool.

Hardware

> **Tip**
>
> The hardware is identical for receiver and transmitter, since the HC-12 can do both.



Pico pin	HC-12 pin	Notes
GP3	SET	OPTIONAL (not implemented on receiver)

Pico pin	HC-12 pin	Notes
GP4 (UART1 TX)	RXD	
GP5 (UART1 RX)	TXD	
3V3	VCC	HC-12 is 3.3 V tolerant
GND	GND	

The HC-12 must be set to the same UART baud as `BAUD` in the script (factory default 9600). To change the air baud, use the firmware shell on the transmitting side:

```
telemetry hc12 baud 19200
```

This persists the new baud to the HC-12's internal flash. Both the transmitter HC-12 and the receiver HC-12 must be reconfigured for the link to work.

Wire format

Frames sent by the firmware look like this:

```
[A5 5A] [type] [payload_len] [payload...] [crc16_le]
```

Field	Size	Purpose
Magic	2 B	Lets the receiver re-sync after dropped bytes (`0xA5 0x5A`).
Type	1 B	Frame kind. Currently only `0x01` (SM update) is defined.
Length	1 B	Length of the payload in bytes.
Payload	N B	Type-specific body (see below).

Field	Size	Purpose
CRC	2 B	CRC-16/CCITT, reflected, init `0xFFFF`, over `[type ‖ len ‖ payload]`. Little-endian.

The SM-update payload (`type = 0x01`, 64 B) matches `struct hc12_sm_update_payload`: a millisecond timestamp, the flight state enum, an armed flag, a reserved field, then altitude / total acceleration / vertical acceleration / vertical velocity, then three orientation angles (yaw, pitch, roll). All floats are 64-bit little-endian.

Why the parser is structured this way

A naïve receive loop calls `print()` for each frame and reads bytes synchronously. This appears to work at high telemetry rates but fails at lower rates because the UART hardware FIFO on the RP2040 is only 32 bytes and overruns whenever `print()` blocks. The script avoids that problem in three ways:

1. **Driver-side buffering**: the UART is opened with `rxbuf=2048`, so bytes keep arriving while Python is busy.
2. **Allocation-free parsing**: one preallocated `bytearray` holds the in-flight bytes; parsing walks it with index arithmetic and re-uses the buffer across iterations.
3. **Deferred output**: decoded lines are queued and only flushed when the UART has no pending bytes. The slow REPL `print()` never blocks the parser.

The CRC is computed via a precomputed 256-entry lookup table, which is roughly an order of magnitude faster than the bit-by-bit reference implementation.

Usage

> **Tip**
>
> If you only want to build a simple HC-12 telemetry tester on a bench-target, you can do so. An example for the pico2w bench build is described in Building the Pico 2 W bench target.

1. Flash MicroPython onto the Pico (micropython.org/download).
2. Copy `rec_zephyr.py` to the Pico as `main.py` (so it runs at boot), for example with `mpremote`:

```
mpremote cp tools/rec_zephyr.py :main.py
```

3. Open the REPL (`mpremote` or any serial terminal at 115200 baud on the Pico's USB CDC). One line should print per received telemetry frame:

   ```
   [OK ] SM t=12345 ms  state=ARMED     armed=1  alt=+0.12
   a=+9.81  ...
   ```

   Frames whose CRC fails appear as `[BAD] type=0x?? len=N`: occasional bad frames are expected at long range or with weak antennas, but a steady stream of them indicates the UART baud is wrong on one side.

Here is an example output of the first successfull MULTIMETER flight:

```
[OK ] SM t=18740 ms  state=ARMED     armed=1  alt=-0.05  a=+10.03
av=-0.01  v=-0.11  ypr=-3.56/+2.90/-1.22
[OK ] SM t=18843 ms  state=ARMED     armed=1  alt=-0.01  a=+10.01
av=-0.05  v=-0.06  ypr=-4.45/+3.23/-1.23
[OK ] SM t=18945 ms  state=ARMED     armed=1  alt=-0.03  a=+10.62
av=+0.58  v=-0.08  ypr=-2.90/+2.43/-1.24
[OK ] SM t=19047 ms  state=ARMED     armed=1  alt=-0.02  a=+15.64
av=+5.61  v=+0.07  ypr=-2.25/+1.68/-1.80
[OK ] SM t=19150 ms  state=ARMED     armed=1  alt=+0.03  a=+37.58
av=+27.49  v=+1.66  ypr=-2.51/+2.82/-5.80
[OK ] SM t=19252 ms  state=ARMED     armed=1  alt=+0.31  a=+50.87
av=+40.86  v=+5.52  ypr=-0.16/+0.08/-8.97
[OK ] SM t=19355 ms  state=ARMED     armed=1  alt=+1.12  a=+35.49
av=+25.44  v=+8.99  ypr=+1.02/+2.13/-11.88
[OK ] SM t=19456 ms  state=ARMED     armed=1  alt=+2.24  a=+28.87
av=+18.66  v=+11.21  ypr=+2.39/+0.72/-14.61
[OK ] SM t=19559 ms  state=ARMED     armed=1  alt=+3.54  a=+25.89
av=+15.31  v=+13.01  ypr=+1.40/-1.42/-12.33
[OK ] SM t=19662 ms  state=ARMED     armed=1  alt=+4.88  a=+22.46
av=+11.72  v=+14.29  ypr=-0.73/-1.17/-6.28
[OK ] SM t=19764 ms  state=ARMED     armed=1  alt=+6.48  a=+21.55
av=+10.86  v=+15.50  ypr=-1.86/+0.66/+1.61
[OK ] SM t=19867 ms  state=ARMED     armed=1  alt=+8.19  a=+21.81
av=+11.34  v=+16.71  ypr=-1.11/+4.88/+10.68
[OK ] SM t=19968 ms  state=BOOST     armed=1  alt=+10.00  a=+21.84
av=+11.71  v=+17.94  ypr=+3.00/+3.72/+19.59
[OK ] SM t=20070 ms  state=BOOST     armed=1  alt=+11.87  a=+21.33
av=+10.50  v=+19.13  ypr=+2.45/-4.48/+31.19
[OK ] SM t=20174 ms  state=BOOST     armed=1  alt=+13.93  a=+20.76
av=+9.03  v=+20.19  ypr=-5.45/+0.74/+44.32
[OK ] SM t=20275 ms  state=BOOST     armed=1  alt=+16.00  a=+20.41
av=+9.46  v=+21.08  ypr=+0.05/+5.76/+57.49
[OK ] SM t=20378 ms  state=BOOST     armed=1  alt=+18.10  a=+20.20
av=+9.99  v=+21.98  ypr=+3.91/+0.03/+71.50
```

```
[OK ] SM t=20480 ms  state=BURNOUT   armed=1  alt=+20.33  a=+1.97
av=-9.51  v=+22.06  ypr=-71.02/-50.08/+87.94
[OK ] SM t=20582 ms  state=BURNOUT   armed=1  alt=+22.47  a=+1.43
av=-11.00  v=+20.88  ypr=-138.33/+11.62/+104.11
[OK ] SM t=20685 ms  state=BURNOUT   armed=1  alt=+24.42  a=+2.04
av=-10.81  v=+19.60  ypr=+120.82/+16.06/+119.71
[OK ] SM t=20787 ms  state=BURNOUT   armed=1  alt=+26.32  a=+1.42
av=-10.17  v=+18.50  ypr=+140.04/-49.49/+136.49
[OK ] SM t=20889 ms  state=BURNOUT   armed=1  alt=+28.01  a=+1.10
av=-10.48  v=+17.29  ypr=-130.06/+0.50/+153.48
[OK ] SM t=20992 ms  state=BURNOUT   armed=1  alt=+29.66  a=+0.85
av=-10.43  v=+16.15  ypr=-135.92/+9.07/+169.21
[OK ] SM t=21094 ms  state=BURNOUT   armed=1  alt=+31.16  a=+0.71
av=-10.69  v=+14.96  ypr=-179.08/+32.83/-176.29
[OK ] SM t=21197 ms  state=BURNOUT   armed=1  alt=+32.52  a=+0.63
av=-10.65  v=+13.72  ypr=+140.31/-21.45/-161.97
[OK ] SM t=21299 ms  state=BURNOUT   armed=1  alt=+33.77  a=+0.61
av=-10.30  v=+12.56  ypr=-128.05/-23.99/-148.44
[OK ] SM t=21401 ms  state=BURNOUT   armed=1  alt=+34.88  a=+0.90
av=-10.16  v=+11.39  ypr=-109.98/+3.06/-135.81
[OK ] SM t=21504 ms  state=BURNOUT   armed=1  alt=+35.88  a=+0.92
av=-10.16  v=+10.21  ypr=-108.25/-2.99/-124.20
[OK ] SM t=21606 ms  state=BURNOUT   armed=1  alt=+36.90  a=+0.34
av=-10.72  v=+9.19  ypr=+164.36/-42.75/-113.24
[OK ] SM t=21709 ms  state=BURNOUT   armed=1  alt=+37.78  a=+0.39
av=-10.89  v=+8.06  ypr=+119.36/-24.85/-102.71
[OK ] SM t=21811 ms  state=BURNOUT   armed=1  alt=+38.60  a=+0.20
av=-10.58  v=+7.02  ypr=-125.22/-10.88/-92.64
[OK ] SM t=21913 ms  state=BURNOUT   armed=1  alt=+39.28  a=+0.80
av=-10.12  v=+5.99  ypr=-100.30/+36.70/-82.96
[OK ] SM t=22017 ms  state=BURNOUT   armed=1  alt=+39.88  a=+0.64
av=-10.24  v=+4.96  ypr=-95.44/+19.17/-72.87
[OK ] SM t=22118 ms  state=BURNOUT   armed=1  alt=+40.35  a=+0.47
av=-10.35  v=+3.95  ypr=-101.31/+33.50/-62.83
[OK ] SM t=22220 ms  state=BURNOUT   armed=1  alt=+40.74  a=+0.58
av=-10.18  v=+2.95  ypr=-94.90/+54.91/-54.07
[OK ] SM t=22322 ms  state=BURNOUT   armed=1  alt=+41.04  a=+0.61
av=-10.14  v=+1.96  ypr=-104.04/+67.25/-46.65
[OK ] SM t=22425 ms  state=BURNOUT   armed=1  alt=+41.27  a=+0.45
av=-10.27  v=+1.01  ypr=+105.94/+81.01/-39.79
[OK ] SM t=22528 ms  state=BURNOUT   armed=1  alt=+41.44  a=+0.33
av=-10.30  v=+0.08  ypr=-100.30/+71.28/-33.34
[OK ] SM t=22629 ms  state=BURNOUT   armed=1  alt=+41.46  a=+0.38
av=-10.19  v=-0.89  ypr=-90.00/+77.01/-27.10
[OK ] SM t=22732 ms  state=MAIN      armed=1  alt=+41.25  a=+0.72
av=-9.97  v=-2.01  ypr=-121.37/+50.38/-22.15
[OK ] SM t=22834 ms  state=MAIN      armed=1  alt=+40.91  a=+0.39
av=-10.11  v=-3.14  ypr=-45.00/+88.02/-19.37
[OK ] SM t=22937 ms  state=MAIN      armed=1  alt=+40.41  a=+1.24
av=-9.62  v=-4.26  ypr=-43.60/+63.43/-15.98
[OK ] SM t=23040 ms  state=MAIN      armed=1  alt=+39.72  a=+1.15
av=-9.36  v=-5.45  ypr=-122.47/+83.75/-12.21
```

```
[OK ] SM t=23141 ms  state=MAIN      armed=1  alt=+38.89  a=+1.45
av=-8.85  v=-6.62  ypr=+120.47/+67.09/-7.60
[OK ] SM t=23244 ms  state=MAIN      armed=1  alt=+38.09  a=+1.06
av=-9.09  v=-7.62  ypr=+135.43/+31.66/-1.29
[OK ] SM t=23347 ms  state=MAIN      armed=1  alt=+37.15  a=+1.37
av=-8.66  v=-8.66  ypr=+129.69/+24.78/+7.51
[OK ] SM t=23448 ms  state=MAIN      armed=1  alt=+36.12  a=+3.37
av=-6.77  v=-9.52  ypr=+129.98/+42.77/+11.86
[OK ] SM t=23552 ms  state=MAIN      armed=1  alt=+35.05  a=+6.12
av=-4.18  v=-10.16  ypr=+156.14/+50.40/+13.17
[OK ] SM t=23653 ms  state=MAIN      armed=1  alt=+34.04  a=+5.00
av=-5.24  v=-10.43  ypr=+146.69/+49.61/+23.01
[OK ] SM t=23755 ms  state=MAIN      armed=1  alt=+33.30  a=+11.51
av=+0.84  v=-10.37  ypr=+115.04/+38.72/+55.42
[OK ] SM t=23859 ms  state=MAIN      armed=1  alt=+33.78  a=+23.17
av=+9.19  v=-8.52  ypr=+100.31/+5.78/+93.51
[OK ] SM t=23960 ms  state=MAIN      armed=1  alt=+34.23  a=+29.01
av=+7.87  v=-7.08  ypr=+53.13/-29.36/+117.47
[OK ] SM t=24062 ms  state=MAIN      armed=1  alt=+34.02  a=+34.33
av=+20.09  v=-4.37  ypr=-79.12/-2.73/+98.43
[OK ] SM t=24165 ms  state=MAIN      armed=1  alt=+33.73  a=+16.94
av=+2.15  v=-3.90  ypr=-29.83/+37.86/+2.75
[OK ] SM t=24267 ms  state=MAIN      armed=1  alt=+33.57  a=+18.95
av=+7.14  v=-2.76  ypr=-72.39/-27.93/-107.93
[OK ] SM t=24370 ms  state=MAIN      armed=1  alt=+33.32  a=+9.14
av=-2.01  v=-2.70  ypr=+69.35/-38.67/+143.15
[OK ] SM t=24472 ms  state=MAIN      armed=1  alt=+32.83  a=+18.48
av=+4.60  v=-3.06  ypr=-58.21/-25.78/+88.17
[OK ] SM t=24575 ms  state=MAIN      armed=1  alt=+32.45  a=+14.15
av=+3.93  v=-2.58  ypr=-2.67/-62.32/+32.50
[OK ] SM t=24677 ms  state=REDUNDANT armed=1  alt=+32.07  a=+12.30
av=-0.23  v=-2.61  ypr=-44.31/+1.78/+7.34
[OK ] SM t=24779 ms  state=REDUNDANT armed=1  alt=+31.58  a=+9.06
av=-2.23  v=-2.92  ypr=-39.51/-10.41/-52.86
[OK ] SM t=24882 ms  state=REDUNDANT armed=1  alt=+31.05  a=+8.83
av=-1.12  v=-3.37  ypr=+11.07/-0.93/-129.73
[OK ] SM t=24984 ms  state=REDUNDANT armed=1  alt=+30.63  a=+11.13
av=+0.58  v=-3.48  ypr=-26.86/+27.24/+167.68
[OK ] SM t=25086 ms  state=REDUNDANT armed=1  alt=+30.25  a=+9.56
av=-0.72  v=-3.52  ypr=-26.51/-21.80/+82.07
[OK ] SM t=25190 ms  state=REDUNDANT armed=1  alt=+29.84  a=+9.03
av=-0.76  v=-3.64  ypr=+26.03/-30.38/+4.41
[OK ] SM t=25291 ms  state=REDUNDANT armed=1  alt=+29.42  a=+10.74
av=+0.47  v=-3.69  ypr=-5.42/-38.53/-38.79
[OK ] SM t=25394 ms  state=REDUNDANT armed=1  alt=+28.95  a=+11.68
av=+0.69  v=-3.68  ypr=-25.77/-37.32/-51.50
[OK ] SM t=25495 ms  state=REDUNDANT armed=1  alt=+28.50  a=+10.57
av=+0.43  v=-3.76  ypr=-8.89/+4.99/-56.86
[OK ] SM t=25598 ms  state=REDUNDANT armed=1  alt=+28.01  a=+12.87
av=+2.14  v=-3.73  ypr=+21.70/+29.93/-74.80
[OK ] SM t=25702 ms  state=REDUNDANT armed=1  alt=+27.46  a=+11.52
av=+0.43  v=-3.76  ypr=+26.72/+7.02/-97.24
```

```
[OK ] SM t=25803 ms  state=REDUNDANT armed=1  alt=+26.96  a=+10.71
av=+0.02  v=-3.91  ypr=-6.69/-31.53/-90.74
[OK ] SM t=25905 ms  state=REDUNDANT armed=1  alt=+26.49  a=+10.35
av=+0.10  v=-3.98  ypr=-21.27/-2.70/-67.53
[OK ] SM t=26007 ms  state=REDUNDANT armed=1  alt=+25.97  a=+11.91
av=+0.83  v=-4.07  ypr=+27.20/+13.52/-54.07
[OK ] SM t=26110 ms  state=REDUNDANT armed=1  alt=+25.56  a=+10.38
av=-0.24  v=-4.04  ypr=+28.48/-16.06/-17.88
[OK ] SM t=26213 ms  state=REDUNDANT armed=1  alt=+25.13  a=+11.58
av=+0.76  v=-4.02  ypr=-24.11/-1.66/+42.19
[OK ] SM t=26314 ms  state=REDUNDANT armed=1  alt=+24.73  a=+9.42
av=-0.60  v=-4.04  ypr=-0.64/-2.27/+83.98
[OK ] SM t=26417 ms  state=REDUNDANT armed=1  alt=+24.35  a=+12.23
av=+1.31  v=-3.99  ypr=-35.13/-32.25/+130.56
[OK ] SM t=26520 ms  state=REDUNDANT armed=1  alt=+23.86  a=+12.32
av=+0.79  v=-3.94  ypr=-49.66/-13.34/+167.29
[OK ] SM t=26621 ms  state=REDUNDANT armed=1  alt=+23.42  a=+10.14
av=+0.16  v=-4.01  ypr=+16.62/-17.58/+166.20
[OK ] SM t=26725 ms  state=REDUNDANT armed=1  alt=+22.88  a=+10.35
av=+0.63  v=-4.09  ypr=+23.60/-27.53/+178.63
[OK ] SM t=26826 ms  state=REDUNDANT armed=1  alt=+22.47  a=+10.94
av=+0.20  v=-4.06  ypr=-27.43/-9.16/-148.33
[OK ] SM t=26929 ms  state=REDUNDANT armed=1  alt=+22.07  a=+11.03
av=-0.62  v=-4.05  ypr=-34.54/+13.91/-135.98
[OK ] SM t=27032 ms  state=REDUNDANT armed=1  alt=+21.55  a=+9.98
av=+0.12  v=-4.24  ypr=+0.58/-18.45/-154.00
[OK ] SM t=27133 ms  state=REDUNDANT armed=1  alt=+21.06  a=+11.24
av=+1.63  v=-4.18  ypr=+13.57/-32.32/-163.05
[OK ] SM t=27236 ms  state=REDUNDANT armed=1  alt=+20.62  a=+10.33
av=-0.30  v=-4.13  ypr=-23.52/-20.67/-152.69
[OK ] SM t=27338 ms  state=REDUNDANT armed=1  alt=+20.13  a=+10.06
av=-0.56  v=-4.25  ypr=-20.88/-13.03/-156.76
[OK ] SM t=27440 ms  state=REDUNDANT armed=1  alt=+19.74  a=+10.81
av=+1.08  v=-4.19  ypr=+14.52/-23.86/-178.03
[OK ] SM t=27543 ms  state=REDUNDANT armed=1  alt=+19.34  a=+10.51
av=+0.46  v=-4.04  ypr=+4.86/-25.46/-174.56
[OK ] SM t=27645 ms  state=REDUNDANT armed=1  alt=+18.82  a=+10.49
av=-0.86  v=-4.19  ypr=-26.64/-2.25/-144.89
[OK ] SM t=27747 ms  state=REDUNDANT armed=1  alt=+18.29  a=+10.38
av=-0.37  v=-4.38  ypr=-27.28/-16.72/-129.82
[OK ] SM t=27850 ms  state=REDUNDANT armed=1  alt=+17.98  a=+11.64
av=+1.90  v=-4.15  ypr=+6.42/-47.46/-127.63
[OK ] SM t=27952 ms  state=REDUNDANT armed=1  alt=+17.52  a=+10.59
av=+0.48  v=-4.08  ypr=-16.84/-32.60/-103.78
[OK ] SM t=28055 ms  state=REDUNDANT armed=1  alt=+17.07  a=+10.48
av=-0.34  v=-4.11  ypr=-8.98/-0.42/-80.02
[OK ] SM t=28157 ms  state=REDUNDANT armed=1  alt=+16.62  a=+10.15
av=-0.35  v=-4.22  ypr=-4.07/-5.63/-69.58
[OK ] SM t=28259 ms  state=REDUNDANT armed=1  alt=+16.14  a=+11.87
av=+1.43  v=-4.20  ypr=-21.74/-29.99/-63.51
[OK ] SM t=28362 ms  state=REDUNDANT armed=1  alt=+15.77  a=+10.67
av=+0.35  v=-4.02  ypr=-18.91/-34.15/-64.91
```

```
[OK ] SM t=28464 ms  state=REDUNDANT armed=1  alt=+15.27  a=+9.05
av=-0.91  v=-4.19  ypr=+2.27/-24.84/-73.26
[OK ] SM t=28567 ms  state=REDUNDANT armed=1  alt=+14.77  a=+10.39
av=+0.31  v=-4.29  ypr=+5.17/-19.77/-77.38
[OK ] SM t=28669 ms  state=REDUNDANT armed=1  alt=+14.23  a=+10.59
av=-0.36  v=-4.41  ypr=-17.64/-25.31/-69.76
[OK ] SM t=28771 ms  state=REDUNDANT armed=1  alt=+13.73  a=+11.29
av=-0.03  v=-4.50  ypr=-27.48/-28.16/-62.40
[BAD] type=0x01 len=64
[OK ] SM t=28976 ms  state=REDUNDANT armed=1  alt=+12.78  a=+10.02
av=-0.24  v=-4.42  ypr=-9.38/-28.42/-34.69
[OK ] SM t=29079 ms  state=REDUNDANT armed=1  alt=+12.42  a=+9.48
av=-1.36  v=-4.41  ypr=-34.75/-20.19/+2.78
[OK ] SM t=29180 ms  state=REDUNDANT armed=1  alt=+11.86  a=+10.57
av=+0.19  v=-4.67  ypr=-32.01/-26.97/+18.91
[OK ] SM t=29283 ms  state=REDUNDANT armed=1  alt=+11.27  a=+10.94
av=+0.94  v=-4.64  ypr=-8.01/-29.92/+14.32
[OK ] SM t=29386 ms  state=REDUNDANT armed=1  alt=+10.72  a=+11.82
av=+1.79  v=-4.58  ypr=+6.41/-21.76/+6.24
[OK ] SM t=29488 ms  state=REDUNDANT armed=1  alt=+10.16  a=+11.95
av=+1.21  v=-4.55  ypr=-10.63/-28.15/+3.77
[OK ] SM t=29590 ms  state=REDUNDANT armed=1  alt=+9.64  a=+11.57
av=+0.68  v=-4.47  ypr=-13.95/-33.15/+1.90
[OK ] SM t=29692 ms  state=REDUNDANT armed=1  alt=+9.14  a=+9.70
av=-0.47  v=-4.49  ypr=-2.00/-25.36/+3.25
[OK ] SM t=29795 ms  state=REDUNDANT armed=1  alt=+8.63  a=+10.66
av=+0.59  v=-4.54  ypr=-1.70/-20.40/+16.20
[OK ] SM t=29898 ms  state=REDUNDANT armed=1  alt=+8.00  a=+10.97
av=+0.58  v=-4.64  ypr=-14.93/-15.17/+37.43
[OK ] SM t=29999 ms  state=REDUNDANT armed=1  alt=+7.50  a=+10.91
av=+0.78  v=-4.57  ypr=-14.05/-19.96/+48.04
[OK ] SM t=30102 ms  state=REDUNDANT armed=1  alt=+6.89  a=+10.61
av=+0.75  v=-4.67  ypr=-4.15/-24.52/+47.85
[OK ] SM t=30205 ms  state=REDUNDANT armed=1  alt=+6.37  a=+10.69
av=+0.86  v=-4.63  ypr=-0.05/-18.74/+50.64
[OK ] SM t=30306 ms  state=REDUNDANT armed=1  alt=+5.94  a=+10.13
av=+0.05  v=-4.55  ypr=-12.57/-14.50/+54.59
[OK ] SM t=30410 ms  state=REDUNDANT armed=1  alt=+5.45  a=+10.09
av=+0.08  v=-4.56  ypr=-13.20/-23.36/+48.02
[OK ] SM t=30511 ms  state=REDUNDANT armed=1  alt=+5.02  a=+10.25
av=+0.50  v=-4.49  ypr=+3.48/-30.36/+37.39
[OK ] SM t=30614 ms  state=REDUNDANT armed=1  alt=+4.54  a=+10.00
av=+0.22  v=-4.49  ypr=+0.24/-24.06/+43.96
[OK ] SM t=30717 ms  state=REDUNDANT armed=1  alt=+4.00  a=+10.85
av=+0.48  v=-4.53  ypr=-25.02/-11.96/+64.82
[OK ] SM t=30818 ms  state=REDUNDANT armed=1  alt=+3.57  a=+9.99
av=-0.23  v=-4.50  ypr=-26.87/-17.10/+71.09
[OK ] SM t=30921 ms  state=REDUNDANT armed=1  alt=+3.17  a=+11.61
av=+1.83  v=-4.34  ypr=+2.70/-34.79/+59.71
[OK ] SM t=31023 ms  state=REDUNDANT armed=1  alt=+2.72  a=+11.02
av=+1.17  v=-4.14  ypr=-5.31/-26.80/+60.16
[OK ] SM t=31125 ms  state=REDUNDANT armed=1  alt=+2.23  a=+10.20
```

```
av=+0.05  v=-4.13  ypr=-14.63/-14.91/+64.17
[OK ] SM t=31228 ms  state=REDUNDANT armed=1  alt=+1.71  a=+10.52
av=+0.61  v=-4.24  ypr=-4.06/-26.02/+55.70
[OK ] SM t=31330 ms  state=REDUNDANT armed=1  alt=+1.32  a=+11.58
av=+1.76  v=-4.03  ypr=+6.63/-33.11/+53.12
[OK ] SM t=31433 ms  state=REDUNDANT armed=1  alt=+0.87  a=+10.48
av=+0.13  v=-4.01  ypr=-21.17/-18.03/+76.64
[OK ] SM t=31535 ms  state=REDUNDANT armed=1  alt=+0.41  a=+20.39
av=+8.67  v=-3.95  ypr=+14.89/-12.80/+95.44
[OK ] SM t=31637 ms  state=REDUNDANT armed=1  alt=-0.12  a=+29.70
av=-18.36  v=-3.60  ypr=+172.30/+5.49/+101.88
[OK ] SM t=31740 ms  state=REDUNDANT armed=1  alt=-0.34  a=+22.87
av=+10.74  v=-2.73  ypr=-178.36/+6.37/+79.93
[OK ] SM t=31842 ms  state=REDUNDANT armed=1  alt=-0.40  a=+24.61
av=-25.94  v=-3.65  ypr=-177.57/+7.04/+53.45
[OK ] SM t=31944 ms  state=REDUNDANT armed=1  alt=-0.74  a=+20.47
av=-18.11  v=-6.52  ypr=+173.96/+17.27/+95.96
[OK ] SM t=32047 ms  state=REDUNDANT armed=1  alt=-1.08  a=+4.51
av=-13.62  v=-6.36  ypr=+18.12/+11.00/+120.68
[OK ] SM t=32149 ms  state=REDUNDANT armed=1  alt=-1.44  a=+8.82
av=-17.77  v=-8.41  ypr=-94.16/-13.56/+117.25
[OK ] SM t=32252 ms  state=REDUNDANT armed=1  alt=-1.83  a=+7.39
av=-13.74  v=-9.45  ypr=-42.60/-9.62/+111.86
[OK ] SM t=32354 ms  state=REDUNDANT armed=1  alt=-2.21  a=+11.35
av=-14.45  v=-10.34  ypr=-52.12/-11.03/+114.81
[OK ] SM t=32456 ms  state=REDUNDANT armed=1  alt=-2.60  a=+6.35
av=-11.62  v=-10.88  ypr=-57.47/-7.19/+119.10
[OK ] SM t=32560 ms  state=REDUNDANT armed=1  alt=-2.87  a=+6.10
av=-12.74  v=-11.11  ypr=-152.98/+12.05/+120.90
[OK ] SM t=32661 ms  state=REDUNDANT armed=1  alt=-3.20  a=+6.92
av=-11.69  v=-11.07  ypr=-99.19/+9.07/+123.14
[OK ] SM t=32763 ms  state=REDUNDANT armed=1  alt=-3.36  a=+9.97
av=-8.18  v=-10.91  ypr=-107.12/+25.84/+123.31
[OK ] SM t=32866 ms  state=REDUNDANT armed=1  alt=-3.39  a=+10.06
av=-6.86  v=-10.47  ypr=-109.49/+24.15/+123.27
[OK ] SM t=32968 ms  state=REDUNDANT armed=1  alt=-3.46  a=+10.03
av=-5.36  v=-9.96  ypr=-109.29/+24.52/+123.26
[OK ] SM t=33071 ms  state=REDUNDANT armed=1  alt=-3.49  a=+10.01
av=-4.18  v=-9.34  ypr=-109.18/+24.39/+123.25
[OK ] SM t=33172 ms  state=REDUNDANT armed=1  alt=-3.40  a=+10.00
av=-3.27  v=-8.57  ypr=-109.31/+24.29/+123.24
[OK ] SM t=33275 ms  state=REDUNDANT armed=1  alt=-3.32  a=+10.02
av=-2.53  v=-7.81  ypr=-109.23/+24.43/+123.23
[OK ] SM t=33378 ms  state=REDUNDANT armed=1  alt=-3.17  a=+10.01
av=-2.03  v=-7.00  ypr=-109.20/+24.41/+123.22
[OK ] SM t=33480 ms  state=REDUNDANT armed=1  alt=-3.07  a=+10.00
av=-1.67  v=-6.29  ypr=-109.33/+24.37/+123.22
[OK ] SM t=33583 ms  state=REDUNDANT armed=1  alt=-2.80  a=+10.01
av=-1.39  v=-5.45  ypr=-109.20/+24.41/+123.21
[OK ] SM t=33684 ms  state=REDUNDANT armed=1  alt=-2.50  a=+10.01
av=-1.21  v=-4.65  ypr=-109.40/+24.21/+123.20
[OK ] SM t=33787 ms  state=REDUNDANT armed=1  alt=-2.28  a=+10.02
```

```
av=-1.07  v=-4.01  ypr=-109.39/+24.56/+123.19
[OK ] SM t=33890 ms  state=REDUNDANT armed=1  alt=-2.13  a=+10.03
av=-0.98  v=-3.50  ypr=-109.19/+24.24/+123.19
[OK ] SM t=33991 ms  state=REDUNDANT armed=1  alt=-2.02  a=+10.01
av=-0.93  v=-3.09  ypr=-109.18/+24.39/+123.18
[OK ] SM t=34095 ms  state=REDUNDANT armed=1  alt=-1.86  a=+10.02
av=-0.88  v=-2.66  ypr=-109.50/+24.49/+123.18
[OK ] SM t=34196 ms  state=REDUNDANT armed=1  alt=-1.66  a=+10.00
av=-0.87  v=-2.23  ypr=-109.41/+24.48/+123.17
[OK ] SM t=34298 ms  state=REDUNDANT armed=1  alt=-1.49  a=+10.00
av=-0.85  v=-1.89  ypr=-109.22/+24.48/+123.17
[OK ] SM t=34402 ms  state=REDUNDANT armed=1  alt=-1.44  a=+10.04
av=-0.80  v=-1.71  ypr=-109.36/+24.27/+123.17
[OK ] SM t=34503 ms  state=REDUNDANT armed=1  alt=-1.23  a=+10.06
av=-0.77  v=-1.37  ypr=-109.29/+24.87/+123.15
[OK ] SM t=34606 ms  state=REDUNDANT armed=1  alt=-1.20  a=+9.98
av=-0.84  v=-1.27  ypr=-109.28/+24.65/+123.14
[OK ] SM t=34708 ms  state=REDUNDANT armed=1  alt=-1.10  a=+10.02
av=-0.80  v=-1.10  ypr=-109.25/+24.60/+123.13
[OK ] SM t=34810 ms  state=LANDED    armed=1  alt=-0.99  a=+10.02
av=-0.79  v=-0.94  ypr=-109.44/+24.60/+123.12
[OK ] SM t=34913 ms  state=LANDED    armed=1  alt=-0.91  a=+10.01
av=-0.81  v=-0.82  ypr=-109.37/+24.29/+123.13
[OK ] SM t=35015 ms  state=LANDED    armed=1  alt=-0.93  a=+10.03
av=-0.79  v=-0.83  ypr=-109.48/+24.47/+123.11
[OK ] SM t=35117 ms  state=LANDED    armed=1  alt=-0.91  a=+10.02
av=-0.79  v=-0.79  ypr=-109.48/+24.42/+123.11
[OK ] SM t=35220 ms  state=LANDED    armed=1  alt=-0.84  a=+10.01
av=-0.81  v=-0.71  ypr=-109.39/+24.83/+123.09
```

Note

State APOGEE is so short-lived that the 10Hz LoRa rate is not enough to display it.

Building the Pico 2 W bench target

For bench work it is useful to flash a Pico 2 W with a minimal AURORA build that only exercises the telemetry stack. No real IMU, no barometer, no SD card. This is what the `rpi_pico2/rp2350a/m33/w` board target plus the `sim` and `nodisk` snippets provide:

```
west build -p -b rpi_pico2/rp2350a/m33/w -S sim -S nodisk
sensor_board
```

After flashing the resulting `build/zephyr/zephyr.uf2` onto the Pico, attach the HC-12 to UART1 as documented in the overlay. The board will boot, start the simulated flight, and emit telemetry frames over the radio for `rec_zephyr.py` to pick up.

Requirements

- A Raspberry Pi Pico (or Pico W / Pico 2) flashed with MicroPython.
- An HC-12 radio module paired with the transmitter on the same channel and baud.
- No host-side Python dependencies, the REPL is the user interface.

sim_fetch.py

Source: `tools/sim_fetch.py`

Extract a file from a running `native_sim` Zephyr instance.

Background

The Zephyr FUSE bridge (`CONFIG_FUSE_FS_ACCESS`) mounts the simulator's filesystem on the host, but opens files write-only, so `cp` from the FUSE mount fails with `EIO`. `sim_fetch.py` works around this by dumping the file over the shell console using `fs read <path>` and reconstructing the bytes from the hex dump on the host.

The hex dump format emitted by `subsys/fs/shell.c cmd_read` is:

```
File size: <n>
<8 hex digits offset>  <up to 16 "NN " bytes>   <ascii gutter>
...
```

Modes

The script exposes two subcommands:

Mode	Description
`parse`	Parse a previously captured console log and write the bytes of one `fs read` invocation to disk.
`pexpect`	Spawn a `native_sim` binary, drive the shell, and fetch one or more files in a single step. Requires the `pexpect` Python package.

`parse`

```
python3 tools/sim_fetch.py parse -o OUTPUT [-i INPUT] [--expect-size
N]
```

- `-i / --input` — path to captured console log (`-` for stdin, default).
- `-o / --output` — destination file on the host.
- `--expect-size` — fail unless the dump header reports this size.

The last complete dump in the input is chosen; partial or interrupted dumps are skipped.

`pexpect`

```
python3 tools/sim_fetch.py pexpect <zephyr.exe> \
    --fetch REMOTE LOCAL [--fetch REMOTE LOCAL ...] \
    [--await-ready PATTERN] [--pre-command CMD] [--wait-for PATTERN]
\
    [--prompt PROMPT] [--timeout S] [--await-timeout S] \
    [--wait-timeout S] [--fetch-timeout S] [-v]
```

Key options:

- `--fetch REMOTE LOCAL` — repeatable; `REMOTE` is the Zephyr path (e.g. `/RAM:/data/flight_0.influx`), `LOCAL` is the host destination.
- `--await-ready PATTERN` — block until this regex is seen after boot before sending any `--pre-command`. Useful for asynchronous startup such as attitude calibration.
- `--pre-command CMD` — shell command(s) to send before fetching (repeatable).
- `--wait-for PATTERN` — after each `--pre-command`, wait for this regex instead of the shell prompt (e.g. for a simulator that keeps streaming).
- The `--*-timeout` options control each phase independently.

After all fetches the script issues `kernel reboot cold` and closes the child.

Example

After building the sensor board for the simulator (`west build -p -b native_sim sensor_board`):

```
python3 ./tools/sim_fetch.py pexpect \
    build/zephyr/zephyr.exe \
```

```
    --await-ready "Attitude calibrated" \
    --pre-command "sim launch" \
    --wait-for "LANDED" \
    --wait-timeout 600 \
    --fetch /RAM:/data/flight_0.influx flight_logs/sim_fetch/
flights.influx \
    --fetch /RAM:/state/audit.0       flight_logs/sim_fetch/
state_audit \
    -v
```

Requirements

- Python 3.10+
- `pexpect` (only for the `pexpect` mode): `pip install pexpect`

sim_flight_kalman.py

Source: `tools/sim_flight_kalman.py`

Run a synthetic rocket flight through a Python mirror of the AURORA Kalman filter and plot the result.

What it does

The script is a 1:1 Python port of the firmware filter pipeline. The two mirrored components are:

- The two-state (altitude, velocity) constant-acceleration Kalman filter from `aurora/lib/filter/kalman.c` (`Filter`, `filter_init`, `filter_predict`, `filter_update`, `filter_detect_apogee`).
- The body-frame gravity tracker from `aurora/lib/sensor/attitude.c` (`Attitude`, `attitude_init`, `attitude_calibrate_sample`, `attitude_calibrate_finish`, `attitude_update`, `attitude_is_calibrated`).

Function names, struct layout, return semantics (0 = OK, 1 = gated/apogee, negative `errno` on error) and arithmetic order all track the C, so any tuning change made in Kconfig or in the C sources can be reproduced in Python by updating the mirrored constants at the top of the file.

The script generates its own synthetic data:

- A pressure signal from the ISA barometric formula.
- A body-frame IMU stream (accel + gyro) with a slow pitch-over after apogee, so the gravity tracker actually has to do something.

- MS5607-flavoured Gaussian noise on top of the pressure stream.

These streams are fed through the filter mirrors at 50 Hz and the results are handed to `plot_flight_data.py`, which produces the seven-panel plot shown below.

> **Note**
>
> Plotting is delegated to `plot_flight_data` — `sim_flight_kalman.py` no longer plots anything itself. To plot recorded telemetry from a flight log, use `plot_flight_data.py --flight DIR` directly.

Usage

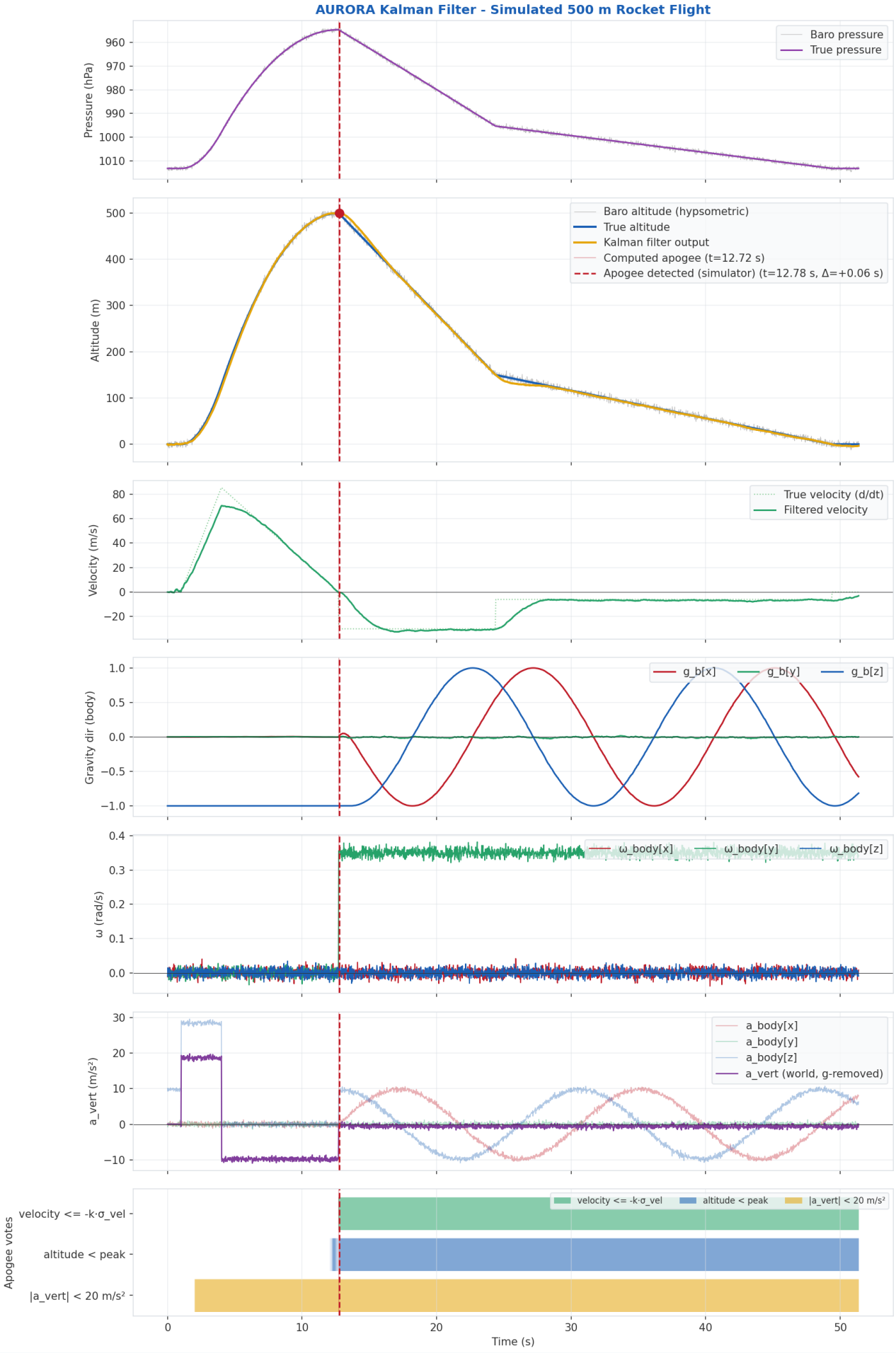
```
python3 tools/sim_flight_kalman.py [--theme {light,dark,both}]
                                   [--show] [--title TITLE]
```

- `--theme {light,dark,both}` — render plots matching the Furo Sphinx docs theme (default `both`).
- `--show` — open the plots in an interactive matplotlib window in addition to writing them to disk.
- `--title TITLE` — override the plot title.

Filter and attitude tuning is not exposed on the CLI on purpose: the constants at the top of the script (`CONFIG_FILTER_*`, `CONFIG_IMU_UP_AXIS_*`) are the same Kconfig defaults the firmware uses, so a default run is directly comparable to a default flight build. To sweep tuning, edit those constants or use `sweep_apogee.py`.

The output file is `flight_simulation.png` (or `_light` / `_dark` suffixed when `--theme both` is used). A typical run looks like this:

```
Ground ref pressure: 101326 Pa (1013.3 hPa)
True apogee:         500.0 m (t = 12.71 s)
Filter apogee:       499.7 m (t = 12.78 s)
Attitude g_mag:      9.794 m/s²
Gated baro updates:  0
[light] Plot saved to flight_simulation.png
```


AURORA Kalman Filter - Simulated 500 m Rocket Flight
Pressure (hPa)
Baro pressure
True pressure
Altitude (m)
Baro altitude (hypsometric)
True altitude
Kalman filter output
Computed apogee (t=12.72 s)
Apogee detected (simulator) (t=12.78 s, Δ=+0.06 s)
Velocity (m/s)
True velocity (d/dt)
Filtered velocity
Gravity dir (body)
g_b[x]
g_b[y]
g_b[z]
ω (rad/s)
ω_body[x]
ω_body[y]
ω_body[z]
a_vert (m/s²)
a_body[x]
a_body[y]
a_body[z]
a_vert (world, g-removed)
Apogee votes
velocity <= -k·σ_vel
altitude < peak
|a_vert| < 20 m/s²
Time (s)

Reuse from other scripts

The filter and attitude mirrors are exposed as plain module-level functions so they can be driven from any other Python tool that wants to replay or sweep them:

- `Filter`, `filter_init`, `filter_predict`, `filter_update`, `filter_detect_apogee`, `filter_votes`
- `Attitude`, `attitude_init`, `attitude_calibrate_sample`, `attitude_calibrate_finish`, `attitude_update`, `attitude_is_calibrated`
- `simulate_flight(...)` — generate the synthetic trajectory used by the default run.

Helpers for loading recorded telemetry (`parse_influx`, `parse_state_audit`, `segment_flights`, etc.) now live in `plot_flight_data.py`.

Requirements

- Python 3.10+
- `numpy`, `matplotlib`

sweep_apogee.py

Source: `tools/sweep_apogee.py`

Grid-search Kalman filter tuning parameters against recorded flights.

What it does

`sweep_apogee.py` iterates over a small grid of `(q_vel, r_meas, debounce)` values and, for each combination, runs every recorded flight in the target directory through `process_real_flight()` from `sim_flight_kalman.py`. For each combination it records, per flight, the signed delta between the filter-detected apogee and the ground-truth apogee computed from the log.

A configuration is rejected if:

- the filter fails to detect an apogee on any flight, or
- the filter fires more than 50 ms *before* the ground-truth apogee on any flight (no early fires).

Surviving configurations are sorted by worst-case lag across flights and the top 15 are printed.

Usage

The script takes no command-line arguments — the flight directory and search grid are currently hard-coded at the top of the file:

```
FLIGHT_DIR = AURORA_DIR / "flight_logs/2026-04-18"

grid = list(itertools.product(
    [1.5, 2.0, 3.0, 4.0, 6.0],    # q_vel
    [2.0, 4.0, 6.0, 8.0, 12.0],   # r_meas
    [1, 2],                       # debounce
))
```

Edit those two blocks to point at a different flight capture or to widen the search space.

Run with:

```
python3 tools/sweep_apogee.py
```

Expected output:

```
Sweeping 50 configs

Top 15 configs (lowest worst-case lag, no early fires):
 q_vel     r  db    f1_Δ    f2_Δ   worst
   2.0   4.0   2   +0.08   +0.12   +0.12
   ...
```

- `f1_Δ`, `f2_Δ` — per-flight lag in seconds (positive = late, negative = early).
- `worst` — worst-case lag across flights used as the ranking key.

Input layout

The referenced `FLIGHT_DIR` must contain:

- `flights.influx` — telemetry dump (InfluxDB line protocol).
- `state_audit` — state machine transition audit log.

Both can be produced by `sim_fetch.py` or harvested from a real flight.

Requirements

- Python 3.10+

- `numpy`
- Must be runnable from the `aurora/tools` directory so the imports of `sim_flight_kalman` (filter mirrors) and `plot_flight_data` (`parse_influx`, `parse_state_audit`, `segment_flights`) resolve.

sync_defconfig.py

Source: `tools/sync_defconfig.py`

Merge the Zephyr-generated `defconfig` into a board-specific `.conf`.

What it does

After a Zephyr build, `build/zephyr/kconfig/defconfig` holds the fully resolved Kconfig state for the chosen board. `sync_defconfig.py` folds that snapshot back into the tracked `boards/<board>.conf` without losing any manual overrides and without re-stating anything that is already covered by the project-wide `prj.conf`.

The merge walks the generated `defconfig` line by line and produces a merged `<project>/boards/<board>.conf` that:

- follows `defconfig`'s ordering and structure;
- drops any entry already covered by `<project>/prj.conf` (overhead cleanup);
- keeps the existing `board.conf` value for keys that are already set there (manual overrides win);
- fills in everything else from `defconfig`;
- appends any `board.conf`-only entries (not in `defconfig` and not in `prj.conf`) at the end.

Blank lines and comments from `defconfig` are preserved so the output keeps the same visual grouping.

Usage

```
python3 tools/sync_defconfig.py <board> <project>
```

Arguments:

- `board` — Zephyr board name (e.g. `sensor_board_v2/rp2040`). Slashes are translated to underscores when resolving the `.conf` path.

- `project` — AURORA project directory (e.g. `sensor_board`).

Expected paths (relative to the `aurora/` root):

- `build/zephyr/kconfig/defconfig` — must exist from a prior `west build`.
- `<project>/prj.conf` — must exist.
- `<project>/boards/<board>.conf` — written (created if missing).

Example:

```
west build -b sensor_board_v2/rp2040 aurora/sensor_board
python3 aurora/tools/sync_defconfig.py sensor_board_v2/rp2040
sensor_board
```

The script reports a summary on stdout:

```
Written aurora/sensor_board/boards/sensor_board_v2_rp2040.conf
  42 kept, 7 added from defconfig, 3 removed (overhead in prj.conf)
```

Requirements

- Python 3.10+
- A completed Zephyr build for the target board.

Overview

Script	Purpose
`gen_flight_replay.py`	Convert a recorded `flights.csv` into a generated C source file consumed by the `fake_sensors` replay backend.
`pad_link_central_example.py`	Reference BLE central for the pad-link library — scans, connects, and prints SM state and computed kinematics from a rocket.
`plot_flight_data.py`	

Script	Purpose
	Plot recorded or simulated flight data — standalone CLI for telemetry logs, importable plotting module for `sim_flight_kalman`.
`rec_zephyr.py`	MicroPython ground-station receiver for HC-12 telemetry — runs on a Raspberry Pi Pico and prints decoded state-machine frames to the REPL.
`sim_fetch.py`	Pull files out of a running `native_sim` Zephyr instance via the shell console.
`sim_flight_kalman.py`	Run a synthetic flight through a Python mirror of the AURORA Kalman filter and attitude tracker.
`sweep_apogee.py`	Grid-search Kalman filter tuning parameters against recorded flights.
`sync_defconfig.py`	Merge the generated Zephyr `defconfig` into a board-specific `.conf`.

Supported Boards

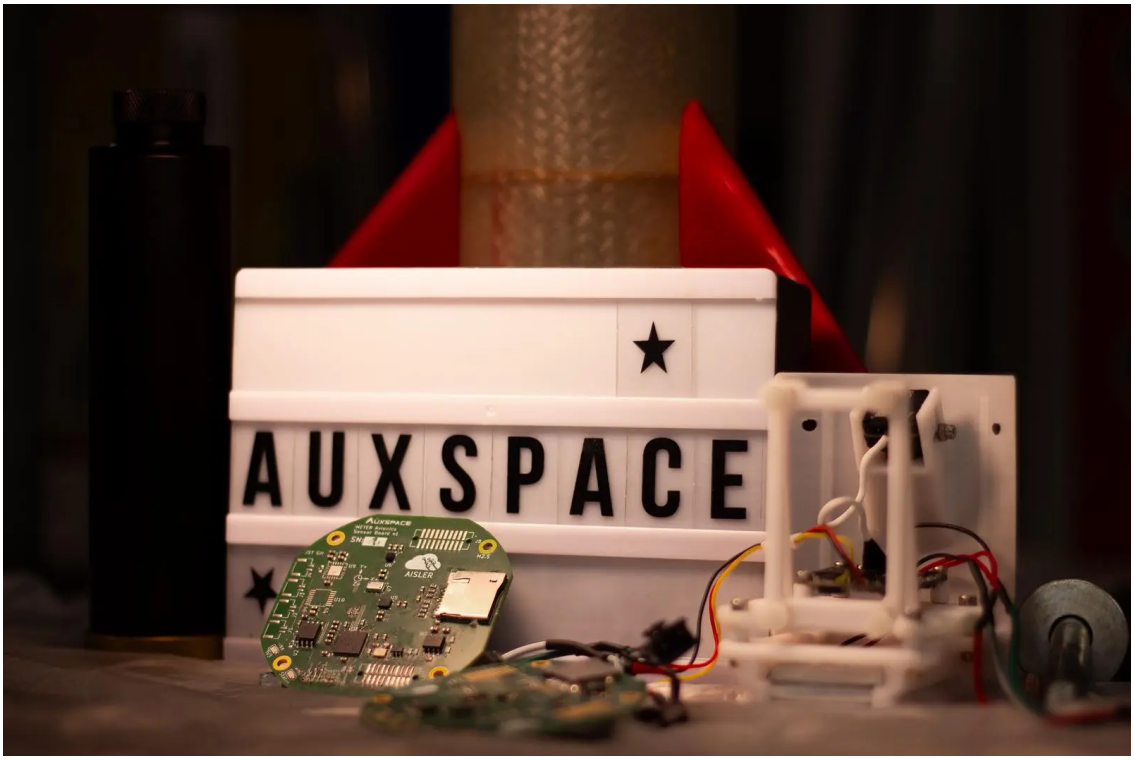
This page lists all boards currently supported by AURORA.

Flight boards

Custom Auxspace PCBs intended for use in flight.

Sensor Board V2

Board Overview



Sensor Board V2

Name:

`sensor_board_v2`

Vendor:

Auxspace e.V.

Status:

Maintained

Architecture:

riscv, arm

SoC:

rp2040, rp2350a

Overview

The Auxspace e.V. sensor_board_v2 PCB is built around the idea to support interchangeable hardware in a stackable rocket avionics system. This version contains a Raspberry Pi Pico with the RP2040 dual core Arm Cortex-M0+ or a Raspberry Pi Pico 2 with the RP2350 dual core RISC-V Hazard3.

Hardware

Raspberry Pi Pico variant:

- Dual core Arm Cortex-M0+ processor running up to 133MHz
- 264KB on-chip SRAM
- 2MB on-board QSPI flash with XIP capabilities
- 26 GPIO pins
- 3 Analog inputs
- 2 UART peripherals
- 2 SPI controllers
- 2 I2C controllers
- 16 PWM channels
- USB 1.1 controller (host/device)
- 8 Programmable I/O (PIO) for custom peripherals
- On-board LED
- 1 Watchdog timer peripheral
- Infineon CYW43439 2.4 GHz Wi-Fi chip (Pico W only)

Raspberry Pi Pico 2 variant:

- Dual Cortex-M33 or Hazard3 processors at up to 150MHz
- 520KB of SRAM, and 4MB of on-board flash memory
- USB 1.1 with device and host support
- Low-power sleep and dormant modes
- Drag-and-drop programming using mass storage over USB
- 26 multi-function GPIO pins including 3 that can be used for ADC
- 2 SPI, 2 I2C, 2 UART, 3 12-bit 500ksps Analogue to Digital - Converter (ADC), 24 controllable PWM channels

- 2 Timer with 4 alarms, 1 AON Timer
- Temperature sensor
- 3 Programmable IO (PIO) blocks, 12 state machines total for custom peripheral support
- Infineon CYW43439 2.4 GHz Wi-Fi chip (Pico 2W only)
  - Flexible, user-programmable high-speed IO
  - Can emulate interfaces such as SD Card and VGA

Supported Features

The `sensor_board_v2` board supports the hardware features listed below.

on-chip / **on-board**

Feature integrated in the SoC / present on the board.

2 / 2

Number of instances that are enabled / disabled.
Click on the label to see the first instance of this feature in the board/SoC DTS files.

`vnd,foo`

Compatible string for the Devicetree binding matching the feature.
Click on the link to view the binding documentation.

`sensor_board_v2/rp2040` target

On-target memory for this board target: 264 KiB of RAM, 2 MiB of Flash.

Type	Location	Description	Compatible
CPU	**on-chip**	ARM Cortex-M0+ CPU2	`arm,cortex-m0+`

Type	Location	Description	Compatible
ADC	on-chip	Raspberry Pi Pico ADC1	`raspberrypi,pico-adc`
Clock control	on-chip	Raspberry Pi Pico clock controller node1	`raspberrypi,pico-clock-controller`
	on-chip	The representation of Raspberry Pi Pico's clock112	`raspberrypi,pico-clock`
	on-chip	The representation of Raspberry Pi Pico's PLL2	`raspberrypi,pico-pll`
	on-chip	The representation of Raspberry Pi Pico ring oscillator1	`raspberrypi,pico-rosc`
	on-chip	The representation of Raspberry Pi Pico external oscillator1	`raspberrypi,pico-xosc`
Counter	on-chip	Raspberry Pi Pico timer1	`raspberrypi,pico-timer`
DMA	on-chip	Raspberry Pi Pico DMA1	`raspberrypi,pico-dma`
Flash controller	on-chip	Raspberry Pi Pico flash controller1	`raspberrypi,pico-flash-controller`
GPIO & Headers	on-chip	Raspberry Pi Pico GPIO1	`raspberrypi,pico-gpio`
	on-chip	Raspberry Pi Pico GPIO Port1	`raspberrypi,pico-gpio-port`

Type	Location	Description	Compatible
I2C	on-chip	Raspberry Pi Pico I2C2	`raspberrypi,pico-i2c`
Interrupt controller	on-chip	ARMv6-M NVIC (Nested Vectored Interrupt Controller) controller1	`arm,v6m-nvic`
Mailbox	on-chip	Raspberry Pi Pico interprocessor mailbox2	`raspberrypi,pico-mbox`
Miscellaneous	on-chip	Raspberry Pi Pico PIO2	`raspberrypi,pico-pio`
	on-chip	Raspberry Pi Pico SIO1	`raspberrypi,pico-sio`
MTD	on-chip	Flash node1	`soc-nv-flash`
Pin control	on-chip	Raspberry Pi Pico Pin Controller1	`raspberrypi,pico-pinctrl`
PWM	on-chip	Raspberry Pi Pico PWM1	`raspberrypi,pico-pwm`
Regulator	on-chip	Raspberry Pi Pico core supply regurator1	`raspberrypi,core-supply-regulator`
Reset controller	on-chip	Raspberry Pi Pico Reset Controller1	`raspberrypi,pico-reset`
RTC	on-chip	Raspberry Pi Pico RTC1	`raspberrypi,pico-rtc`

Type	Location	Description	Compatible
Sensors	**on-board**	STMicroelectronics LSM6DSO32 6-axis IMU (Inertial Measurement Unit) sensor accessed through I2C bus1	`st,lsm6dso32`
	on-board	TE Connectivity MS5607 digital pressure and temperature sensor1	`meas,ms5607`
	on-chip	Raspberry Pi Pico family temperature sensor node1	`raspberrypi,pico-temp`
Serial controller	**on-chip**	Raspberry Pi Pico UART2	`raspberrypi,pico-uart`
SPI	**on-chip**	Raspberry Pi Pico SPI2	`raspberrypi,pico-spi`
SRAM	**on-chip**	Generic on-chip SRAM1	`mmio-sram`
Timer	**on-chip**	ARMv6-M System Tick1	`arm,armv6m-systick`
USB	**on-chip**	Raspberry Pi Pico USB Device Controller1	`raspberrypi,pico-usbd`
Watchdog	**on-chip**	Raspberry Pi Pico Watchdog1	`raspberrypi,pico-watchdog`

`sensor_board_v2/rp2040/w` target

On-target memory for this board target: 264 KiB of RAM, 2 MiB of Flash.

Type	Location	Description	Compatible
CPU	on-chip	ARM Cortex-M0+ CPU2	`arm,cortex-m0+`
ADC	on-chip	Raspberry Pi Pico ADC1	`raspberrypi,pico-adc`
Clock control	on-chip	Raspberry Pi Pico clock controller node1	`raspberrypi,pico-clock-controller`
	on-chip	The representation of Raspberry Pi Pico's clock112	`raspberrypi,pico-clock`
	on-chip	The representation of Raspberry Pi Pico's PLL2	`raspberrypi,pico-pll`
	on-chip	The representation of Raspberry Pi Pico ring oscillator1	`raspberrypi,pico-rosc`
	on-chip	The representation of Raspberry Pi Pico external oscillator1	`raspberrypi,pico-xosc`
Counter	on-chip	Raspberry Pi Pico timer1	`raspberrypi,pico-timer`
DMA	on-chip	Raspberry Pi Pico DMA1	`raspberrypi,pico-dma`
Flash controller	on-chip	Raspberry Pi Pico flash controller1	`raspberrypi,pico-flash-controller`
GPIO & Headers	on-chip	Raspberry Pi Pico GPIO1	`raspberrypi,pico-gpio`

Type	Location	Description	Compatible
	on-chip	Raspberry Pi Pico GPIO Port1	`raspberrypi,pico-gpio-port`
	on-board	Infineon CYW43 GPIO controller1	`infineon,cyw43-gpio`
I2C	**on-chip**	Raspberry Pi Pico I2C2	`raspberrypi,pico-i2c`
Interrupt controller	**on-chip**	ARMv6-M NVIC (Nested Vectored Interrupt Controller) controller1	`arm,v6m-nvic`
Mailbox	**on-chip**	Raspberry Pi Pico interprocessor mailbox2	`raspberrypi,pico-mbox`
Miscellaneous	**on-chip**	Raspberry Pi Pico PIO11	`raspberrypi,pico-pio`
	on-chip	Raspberry Pi Pico SIO1	`raspberrypi,pico-sio`
MTD	**on-chip**	Flash node1	`soc-nv-flash`
Pin control	**on-chip**	Raspberry Pi Pico Pin Controller1	`raspberrypi,pico-pinctrl`
PWM	**on-chip**	Raspberry Pi Pico PWM1	`raspberrypi,pico-pwm`
Regulator	**on-chip**	Raspberry Pi Pico core supply regurator1	`raspberrypi,core-supply-regulator`

Type	Location	Description	Compatible
Reset controller	**on-chip**	Raspberry Pi Pico Reset Controller1	`raspberrypi,pico-reset`
RTC	**on-chip**	Raspberry Pi Pico RTC1	`raspberrypi,pico-rtc`
Sensors	**on-board**	STMicroelectronics LSM6DSO32 6-axis IMU (Inertial Measurement Unit) sensor accessed through I2C bus1	`st,lsm6dso32`
	on-board	TE Connectivity MS5607 digital pressure and temperature sensor1	`meas,ms5607`
	on-chip	Raspberry Pi Pico family temperature sensor node1	`raspberrypi,pico-temp`
Serial controller	**on-chip**	Raspberry Pi Pico UART2	`raspberrypi,pico-uart`
SPI	**on-chip**	Raspberry Pi Pico SPI2	`raspberrypi,pico-spi`
	on-board	Raspberry Pi Pico SPI via PIO1	`raspberrypi,pico-spi-pio`
SRAM	**on-chip**	Generic on-chip SRAM1	`mmio-sram`
Timer	**on-chip**	ARMv6-M System Tick1	`arm,armv6m-systick`
USB	**on-chip**	Raspberry Pi Pico USB Device Controller1	`raspberrypi,pico-usbd`

Type	Location	Description	Compatible
Watchdog	**on-chip**	Raspberry Pi Pico Watchdog1	`raspberrypi,pico-watchdog`
Wi-Fi	**on-board**	AIROC Wi-Fi Connectivity over SPI1	`infineon,airoc-wifi`

`sensor_board_v2/rp2350a/hazard3` target

On-target memory for this board target: 520 KiB of RAM, 4 MiB of Flash.

Type	Location	Description	Compatible
CPU	**on-chip**	Generic RISC-V CPU2	`riscv`
ADC	**on-chip**	Raspberry Pi Pico ADC1	`raspberrypi,pico-adc`
Clock control	**on-chip**	The representation of Raspberry Pi Pico's clock112	`raspberrypi,pico-clock`
	on-chip	The representation of Raspberry Pi Pico's PLL2	`raspberrypi,pico-pll`
	on-chip	The representation of Raspberry Pi Pico ring oscillator1	`raspberrypi,pico-rosc`
	on-chip	The representation of Raspberry Pi Pico external oscillator1	`raspberrypi,pico-xosc`
	on-chip	Raspberry Pi Pico clock controller node1	`raspberrypi,pico-clock-controller`

Type	Location	Description	Compatible
Counter	**on-chip**	Raspberry Pi Pico timer2	`raspberrypi,pico-timer`
Cryptographic accelerator	**on-chip**	Raspberry Pi Pico SHA-256 accelerator1	`raspberrypi,pico-sha256`
DMA	**on-chip**	Raspberry Pi Pico DMA1	`raspberrypi,pico-dma`
Flash controller	**on-chip**	Raspberry Pi Pico flash controller1	`raspberrypi,pico-flash-controller`
GPIO & Headers	**on-chip**	Raspberry Pi Pico GPIO1	`raspberrypi,pico-gpio`
	on-chip	Raspberry Pi Pico GPIO Port2	`raspberrypi,pico-gpio-port`
I2C	**on-chip**	Raspberry Pi Pico I2C2	`raspberrypi,pico-i2c`
Interrupt controller	**on-chip**	Hazard3 interrupt controller1	`hazard3,hazard3-intc`
Mailbox	**on-chip**	Raspberry Pi Pico interprocessor mailbox1	`raspberrypi,pico-mbox`
Miscellaneous	**on-chip**	Raspberry Pi Pico PIO3	`raspberrypi,pico-pio`
	on-chip	Raspberry Pi Pico SIO1	`raspberrypi,pico-sio`

Type	Location	Description	Compatible
MTD	**on-chip**	Flash node1	`soc-nv-flash`
	on-board	Fixed partitions of a flash (or other non-volatile storage) memory1	`fixed-partitions`
Pin control	**on-chip**	Raspberry Pi Pico Pin Controller1	`raspberrypi,pico-pinctrl`
PWM	**on-chip**	Raspberry Pi Pico PWM1	`raspberrypi,pico-pwm`
Regulator	**on-chip**	Raspberry Pi Pico core supply regurator1	`raspberrypi,core-supply-regulator`
Reset controller	**on-chip**	Raspberry Pi Pico Reset Controller1	`raspberrypi,pico-reset`
RNG	**on-chip**	Raspberry Pi Pico RNG/Entropy1	`raspberrypi,pico-rng`
Sensors	**on-board**	STMicroelectronics LSM6DSO32 6-axis IMU (Inertial Measurement Unit) sensor accessed through I2C bus1	`st,lsm6dso32`
	on-board	TE Connectivity MS5607 digital pressure and temperature sensor1	`meas,ms5607`
	on-chip	Raspberry Pi Pico family temperature sensor node1	`raspberrypi,pico-temp`
Serial controller	**on-chip**	Raspberry Pi Pico UART2	`raspberrypi,pico-uart`

Type	Location	Description	Compatible
SPI	on-chip	Raspberry Pi Pico SPI2	`raspberrypi,pico-spi`
SRAM	on-chip	Generic on-chip SRAM1	`mmio-sram`
Timer	on-chip	RISC-V Machine Timer1	`riscv,machine-timer`
USB	on-chip	Raspberry Pi Pico USB Device Controller1	`raspberrypi,pico-usbd`
Watchdog	on-chip	Raspberry Pi Pico Watchdog1	`raspberrypi,pico-watchdog`

`sensor_board_v2/rp2350a/m33` target

On-target memory for this board target: 520 KiB of RAM, 4 MiB of Flash.

Type	Location	Description	Compatible
CPU	on-chip	ARM Cortex-M33 CPU2	`arm,cortex-m33`
ADC	on-chip	Raspberry Pi Pico ADC1	`raspberrypi,pico-adc`
Clock control	on-chip	The representation of Raspberry Pi Pico's clock112	`raspberrypi,pico-clock`
	on-chip	The representation of Raspberry Pi Pico's PLL2	`raspberrypi,pico-pll`

Type	Location	Description	Compatible
	on-chip	The representation of Raspberry Pi Pico ring oscillator1	`raspberrypi,pico-rosc`
	on-chip	The representation of Raspberry Pi Pico external oscillator1	`raspberrypi,pico-xosc`
	on-chip	Raspberry Pi Pico clock controller node1	`raspberrypi,pico-clock-controller`
Counter	on-chip	Raspberry Pi Pico timer2	`raspberrypi,pico-timer`
Cryptographic accelerator	on-chip	Raspberry Pi Pico SHA-256 accelerator1	`raspberrypi,pico-sha256`
DMA	on-chip	Raspberry Pi Pico DMA1	`raspberrypi,pico-dma`
Flash controller	on-chip	Raspberry Pi Pico flash controller1	`raspberrypi,pico-flash-controller`
GPIO & Headers	on-chip	Raspberry Pi Pico GPIO1	`raspberrypi,pico-gpio`
	on-chip	Raspberry Pi Pico GPIO Port2	`raspberrypi,pico-gpio-port`
I2C	on-chip	Raspberry Pi Pico I2C2	`raspberrypi,pico-i2c`
Interrupt controller	on-chip	ARMv8-M NVIC (Nested Vectored Interrupt Controller)1	`arm,v8m-nvic`

Type	Location	Description	Compatible
Mailbox	**on-chip**	Raspberry Pi Pico interprocessor mailbox1	`raspberrypi,pico-mbox`
Miscellaneous	**on-chip**	Raspberry Pi Pico PIO3	`raspberrypi,pico-pio`
	on-chip	Raspberry Pi Pico SIO1	`raspberrypi,pico-sio`
MTD	**on-chip**	Flash node1	`soc-nv-flash`
	on-board	Fixed partitions of a flash (or other non-volatile storage) memory1	`fixed-partitions`
Pin control	**on-chip**	Raspberry Pi Pico Pin Controller1	`raspberrypi,pico-pinctrl`
PWM	**on-chip**	Raspberry Pi Pico PWM1	`raspberrypi,pico-pwm`
Regulator	**on-chip**	Raspberry Pi Pico core supply regurator1	`raspberrypi,core-supply-regulator`
Reset controller	**on-chip**	Raspberry Pi Pico Reset Controller1	`raspberrypi,pico-reset`
RNG	**on-chip**	Raspberry Pi Pico RNG/Entropy1	`raspberrypi,pico-rng`
Sensors	**on-board**	STMicroelectronics LSM6DSO32 6-axis IMU (Inertial Measurement Unit) sensor accessed through I2C bus1	`st,lsm6dso32`

Type	Location	Description	Compatible
	on-board	TE Connectivity MS5607 digital pressure and temperature sensor1	`meas,ms5607`
	on-chip	Raspberry Pi Pico family temperature sensor node1	`raspberrypi,pico-temp`
Serial controller	**on-chip**	Raspberry Pi Pico UART2	`raspberrypi,pico-uart`
SPI	**on-chip**	Raspberry Pi Pico SPI2	`raspberrypi,pico-spi`
SRAM	**on-chip**	Generic on-chip SRAM1	`mmio-sram`
Timer	**on-chip**	ARMv8-M System Tick1	`arm,armv8m-systick`
USB	**on-chip**	Raspberry Pi Pico USB Device Controller1	`raspberrypi,pico-usbd`
Watchdog	**on-chip**	Raspberry Pi Pico Watchdog1	`raspberrypi,pico-watchdog`

`sensor_board_v2/rp2350a/m33/w` target

On-target memory for this board target: 520 KiB of RAM, 4 MiB of Flash.

Type	Location	Description	Compatible
CPU	**on-chip**	ARM Cortex-M33 CPU2	`arm,cortex-m33`

Type	Location	Description	Compatible
ADC	on-chip	Raspberry Pi Pico ADC1	`raspberrypi,pico-adc`
Clock control	on-chip	The representation of Raspberry Pi Pico's clock112	`raspberrypi,pico-clock`
	on-chip	The representation of Raspberry Pi Pico's PLL2	`raspberrypi,pico-pll`
	on-chip	The representation of Raspberry Pi Pico ring oscillator1	`raspberrypi,pico-rosc`
	on-chip	The representation of Raspberry Pi Pico external oscillator1	`raspberrypi,pico-xosc`
	on-chip	Raspberry Pi Pico clock controller node1	`raspberrypi,pico-clock-controller`
Counter	on-chip	Raspberry Pi Pico timer2	`raspberrypi,pico-timer`
Cryptographic accelerator	on-chip	Raspberry Pi Pico SHA-256 accelerator1	`raspberrypi,pico-sha256`
DMA	on-chip	Raspberry Pi Pico DMA1	`raspberrypi,pico-dma`
Flash controller	on-chip	Raspberry Pi Pico flash controller1	`raspberrypi,pico-flash-controller`
GPIO & Headers	on-chip	Raspberry Pi Pico GPIO1	`raspberrypi,pico-gpio`

Type	Location	Description	Compatible
	on-chip	Raspberry Pi Pico GPIO Port2	`raspberrypi,pico-gpio-port`
	on-board	Infineon CYW43 GPIO controller1	`infineon,cyw43-gpio`
I2C	on-chip	Raspberry Pi Pico I2C2	`raspberrypi,pico-i2c`
Interrupt controller	on-chip	ARMv8-M NVIC (Nested Vectored Interrupt Controller)1	`arm,v8m-nvic`
Mailbox	on-chip	Raspberry Pi Pico interprocessor mailbox1	`raspberrypi,pico-mbox`
Miscellaneous	on-chip	Raspberry Pi Pico PIO12	`raspberrypi,pico-pio`
	on-chip	Raspberry Pi Pico SIO1	`raspberrypi,pico-sio`
MTD	on-chip	Flash node1	`soc-nv-flash`
	on-board	Fixed partitions of a flash (or other non-volatile storage) memory1	`fixed-partitions`
Pin control	on-chip	Raspberry Pi Pico Pin Controller1	`raspberrypi,pico-pinctrl`
PWM	on-chip	Raspberry Pi Pico PWM1	`raspberrypi,pico-pwm`

Type	Location	Description	Compatible
Regulator	**on-chip**	Raspberry Pi Pico core supply regurator1	`raspberrypi,core-supply-regulator`
Reset controller	**on-chip**	Raspberry Pi Pico Reset Controller1	`raspberrypi,pico-reset`
RNG	**on-chip**	Raspberry Pi Pico RNG/Entropy1	`raspberrypi,pico-rng`
Sensors	**on-board**	STMicroelectronics LSM6DSO32 6-axis IMU (Inertial Measurement Unit) sensor accessed through I2C bus1	`st,lsm6dso32`
	on-board	TE Connectivity MS5607 digital pressure and temperature sensor1	`meas,ms5607`
	on-chip	Raspberry Pi Pico family temperature sensor node1	`raspberrypi,pico-temp`
Serial controller	**on-chip**	Raspberry Pi Pico UART2	`raspberrypi,pico-uart`
SPI	**on-chip**	Raspberry Pi Pico SPI2	`raspberrypi,pico-spi`
	on-board	Raspberry Pi Pico SPI via PIO1	`raspberrypi,pico-spi-pio`
SRAM	**on-chip**	Generic on-chip SRAM1	`mmio-sram`
Timer	**on-chip**	ARMv8-M System Tick1	`arm,armv8m-systick`

Type	Location	Description	Compatible
USB	**on-chip**	Raspberry Pi Pico USB Device Controller[1]	`raspberrypi,pico-usbd`
Watchdog	**on-chip**	Raspberry Pi Pico Watchdog[1]	`raspberrypi,pico-watchdog`
Wi-Fi	**on-board**	AIROC Wi-Fi Connectivity over SPI[1]	`infineon,airoc-wifi`

Connections and IOs

The connectors remain undocumented for the time being, since the PCB is under active development.

Wireless connectivity (`_w` variants)

The `_w` board variants are fitted with the Infineon CYW43439 combo radio (as on the Raspberry Pi Pico W / Pico 2 W). It hangs off a PIO-SPI ("gSPI") bus on GPIO 23/24/25/29, defined in `sensor_board_v2_wifi.dtsi`.

Wi-Fi (supported)

Wi-Fi works through the upstream `infineon,airoc-wifi` driver. Enable it with `CONFIG_WIFI=y` / `CONFIG_WIFI_AIROC=y` plus the module selection (`CONFIG_CYW43439=y`, `CONFIG_CYW43439_MURATA_1YN=y`).

Bluetooth / BLE (**NOT** supported)

> **⚠ Warning**
>
> There is currently **no way to enable Bluetooth on the `_w` boards through configuration alone.** The CYW43439's Bluetooth shares the same PIO-SPI bus as Wi-Fi, and **Zephyr has no HCI transport driver for CYW43439 BT over that shared bus** (verified against the *Zephyr* 4.4 tree).

Why config-only attempts fail:

- The only AIROC Bluetooth HCI driver in Zephyr is `hci_uart_infineon.c` (`compatible = "infineon,bt-hci-uart"`), which requires a **dedicated BT UART** (see the `cy8cproto_062_4343w` board). Every CYW43439 BT Kconfig option (`CONFIG_CYW43439`, …) gates on `BT_H4` (UART). The Pico W does not route BT out on a UART; its BT is multiplexed on the Wi-Fi PIO SPI bus.
- The Zephyr 4.4 migration guide folded the old combo-chip compatible into the UART driver: `infineon,cyw43xxx-bt-hci` => `infineon,bt-hci-uart`, `CONFIG_BT_CYW43XX` => `CONFIG_BT_HCI_UART_INFINEON`.
- Upstream `rpi_pico_rp2040_w.dts` deliberately wires Wi-Fi only — no BT node, no `chosen { zephyr,bt-hci }`.

Symptom of trying anyway: declaring a `bt-hci` / `infineon,airoc-bt-hci` node plus `chosen { zephyr,bt-hci = ... }` makes the BT host `DEVICE_DT_GET()` the node, but no driver has a matching `DT_DRV_COMPAT`, so the link fails with `undefined reference to __device_dts_ord_<N>` (the node number shifts as the DT is restructured, but the cause is always “no driver behind the compatible”).

The only code that drives CYW43439 BT over the shared bus is the Pico SDK’s BTStack transport (`hal_rpi_pico/.../pico_cyw43_driver`), which is a separate host stack and is **not** wired into Zephyr’s Bluetooth subsystem.

Enabling BLE here is therefore a driver-development task (port the BTStack cyw43 transport to a Zephyr `bt-hci` driver with Wi-Fi/BT bus arbitration), not a configuration task.

Programming and Debugging

The `sensor_board_v2` board supports the runners and associated west commands listed below.

	flash	debug	attach	rtt	debugserver	reset
blackmagicprobe						
jlink						
openocd	(default)	(default)				
probe-rs						
pyocd						

	flash	debug	attach	rtt	debugserver	reset
uf2						

> **⚠ Warning**
>
> If no µSD-Card is inserted, the board will have trouble booting, since the SPI-SDHC driver has no way to detect card presence without a card-detect-pin!

Building

RP2040

The aurora software for the RP2040 is built with `sysbuild` since it uses the MCUBoot boot loader:

```
west build -p -b sensor_board_v2/rp2040 --sysbuild sensor_board
```

RP2350

The aurora software for the RP2350 is built without `sysbuild` at the moment.

```
west build -p -b sensor_board_v2/rp2350a/hazard3 sensor_board
```

Flashing

RP2040

Flashing the MCUBoot bootloader can not be done with `west flash` since it does not install both binary files in the correct location in the flash. Instead the software can be flashed with `picotool` allowing flashing via the USB interface. For this the board needs to be put into bootloader mode by holding the `BOOTSEL` button while plugging the board into the computer. Then the following commands can be used to flash the bootloader and the application:

```
# Flash the bootloader
picotool load build/mcuboot/zephyr/zephyr.uf2
# Flash the application
picotool load build/sensor_board/zephyr/zephyr.signed.bin --offset
```

```
0x10010000
# Reboot the board
picotool reboot
```

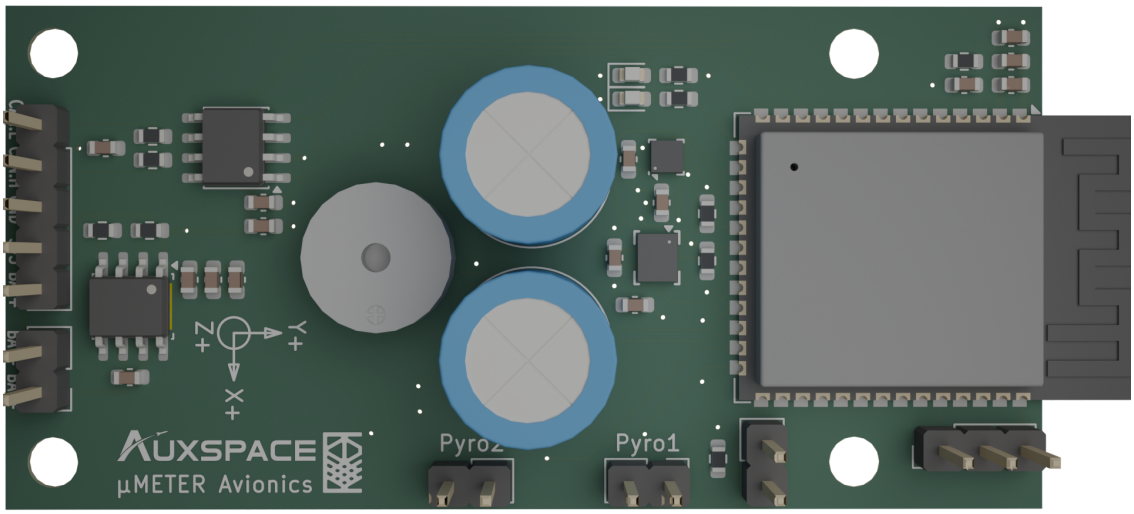
> **Note**
>
> `picotool` can be obtained either by building it from source following the instructions on the picotool repository or by installing it as a prebuilt binary from the release page of the pico-sdk-tools repository.

RP2350

Flashing the RP2350 can be done by copying the generated `zephyr.uf2` located in the `build/sensor_board/zephyr` directory to the board's USB mass storage device. The board needs to be put into bootloader mode by holding the `BOOTSEL` button while plugging the board into the computer.

µMETER

Board Overview



µMETER

Name:

`micrometer`

Vendor:

Auxspace e.V.

Status:

Maintained

Architecture:

xtensa

SoC:

esp32s3

Overview

The Auxspace e.V. µMETER (micrometer) PCB is a minimal flight computer on a 4-layer PCB with an ESP32-S3-WROOM-1 from Espressif.

Hardware

ESP32-S3 is a low-power MCU-based system on a chip (SoC) with integrated 2.4 GHz Wi-Fi and Bluetooth® Low Energy (Bluetooth LE). It consists of high-performance dual-core microprocessor (Xtensa® 32-bit LX7), a low power coprocessor, a Wi-Fi baseband, a Bluetooth LE baseband, RF module, and numerous peripherals. More about that can be found in the zephyr docs of the ESP32-S3.

Specifically added features include:

- 6-DoF IMU
- Barometer
- Buzzer
- 2-channel Pyro ignition
- CAN-bus for communication with posible expansion PCBs
- µSD-Card slot

Revisions

The µMETER PCB exists in two hardware revisions, selected via the Zephyr board revision suffix. Revision 1 is the default, so the bare board name resolves to it.

Target	PCB	Notes
`micrometer/esp32s3/procpu`	v1	default (same as `@1`)
`micrometer@1/esp32s3/procpu`	v1	explicit rev. 1
`micrometer@2/esp32s3/procpu`	v2	updated sensors, dedicated ARM pin

Revision 2 additionally changes to native (4-bit) SDHC instead of the SPI-attached card, uses a faster and better barometer and adds a dedicated “Remove Before Flight” pin for launch-safety.

Supported Features

The `micrometer` board supports the hardware features listed below.

on-chip / **on-board**

Feature integrated in the SoC / present on the board.

2 / 2

Number of instances that are enabled / disabled.
Click on the label to see the first instance of this feature in the board/SoC DTS files.

`vnd,foo`

Compatible string for the Devicetree binding matching the feature.
Click on the link to view the binding documentation.

`micrometer@1/esp32s3/procpu` target

On-target memory for this board target: 416 KiB of RAM, 8 MiB of Flash.

Type	Location	Description	Compatible
CPU	on-chip	Espressif Xtensa LX7 CPU2	`espressif,xtensa-lx7`
ADC	on-chip	ESP32 ADC2	`espressif,esp32-adc`
Bluetooth	on-chip	Bluetooth HCI for Espressif ESP321	`espressif,esp32-bt-hci`
CAN	on-chip	ESP32 Two-Wire Automotive Interface (TWAI)1	`espressif,esp32-twai`
Clock control	on-chip	ESP32 Clock (Power & Clock Controller Module) Module1	`espressif,esp32-clock`
Counter	on-chip	ESP32 Counter Driver based on RTC Main Timer1	`espressif,esp32-rtc-timer`
	on-chip	ESP32 general-purpose timers13	`espressif,esp32-timer`
	on-chip	ESP32 counters4	`espressif,esp32-counter`
Cryptographic accelerator	on-chip	Espressif ESP32 SHA Hardware Accelerator1	`espressif,esp32-sha`
	on-chip	Espressif ESP32 family AES Hardware Accelerator1	`espressif,esp32-aes`
DMA	on-chip	ESP32 GDMA (General Direct Memory Access)1	`espressif,esp32-gdma`

Type	Location	Description	Compatible
Flash controller	**on-chip**	ESP32 flash controller1	`espressif,esp32-flash-controller`
GPIO & Headers	**on-chip**	ESP32 GPIO controller2	`espressif,esp32-gpio`
I2C	**on-chip**	ESP32 I2C2	`espressif,esp32-i2c`
I2S	**on-chip**	ESP32 I2S2	`espressif,esp32-i2s`
Input	**on-chip**	ESP32 touch sensor input1	`espressif,esp32-touch`
	on-board	Group of GPIO-bound input keys1	`gpio-keys`
Interrupt controller	**on-chip**	ESP32 Interrupt controller1	`espressif,esp32-intc`
IPM	**on-chip**	ESP32 soft inter processor message1	`espressif,esp32-ipm`
LED	**on-board**	Group of PWM-controlled LEDs1	`pwm-leds`
Mailbox	**on-chip**	ESP32 soft mailbox1	`espressif,mbox-esp32`
Memory controller	**on-chip**	ESP32 pseudo-static RAM controller1	`espressif,esp32-psram`

Type	Location	Description	Compatible
MIPI-DBI	on-chip	ESP32 LCD-CAM display interface1	`espressif,esp32-lcd-cam-mipi-dbi`
Miscellaneous	on-chip	ESP32 LCD-CAM controller1	`espressif,esp32-lcd-cam`
	on-board	PWM passive buzzer1	`auxspaceev,pwm-buzzer`
MTD	on-chip	Flash node1	`soc-nv-flash`
Pin control	on-chip	ESP32 pin controller1	`espressif,esp32-pinctrl`
PWM	on-chip	ESP32 LED Control (LEDC)1	`espressif,esp32-ledc`
	on-chip	ESP32 Motor Control Pulse Width Modulator (MCPWM)2	`espressif,esp32-mcpwm`
Pyrotechnic Ignition	on-board	Basic pyro module1	`auxspaceev,basic-pyro`
RNG	on-chip	ESP32 TRNG (True Random Number Generator)1	`espressif,esp32-trng`
SDHC	on-chip	ESP32 SDHC controller1	`espressif,esp32-sdhc`
	on-chip	ESP32 SDHC controller slot2	`espressif,esp32-sdhc-slot`

Type	Location	Description	Compatible
Sensors	**on-board**	STMicroelectronics LSM6DSO32 6-axis IMU (Inertial Measurement Unit) sensor accessed through I2C bus1	`st,lsm6dso32`
	on-board	STMicroelectronics LPS22HH pressure and temperature sensor connected to I2C bus1	`st,lps22hh`
	on-chip	ESP32 internal temperature sensor1	`espressif,esp32-temp`
	on-chip	ESP32 Pulse Counter (PCNT)1	`espressif,esp32-pcnt`
Serial controller	**on-chip**	ESP32 UART3	`espressif,esp32-uart`
	on-chip	ESP32 UART1	`espressif,esp32-usb-serial`
SPI	**on-chip**	ESP32 SPI controller2	`espressif,esp32-spi`
SRAM	**on-chip**	Generic on-chip SRAM2	`mmio-sram`
USB	**on-chip**	Espressif USB-OTG full-speed controller (DWC2 compatible) with the internal FS/LS PHY1	`espressif,esp32-usb-otg-fs`
Video	**on-chip**	ESP32 LCD-CAM camera interface1	`espressif,esp32-lcd-cam-dvp`

Type	Location	Description	Compatible
Watchdog	on-chip	ESP32 XT Watchdog Timer1	`espressif,esp32-xt-wdt`
	on-chip	ESP32 watchdog2	`espressif,esp32-watchdog`
Wi-Fi	on-chip	ESP32 SoC Wi-Fi1	`espressif,esp32-wifi`

`micrometer@2/esp32s3/procpu` target

On-target memory for this board target: 416 KiB of RAM, 8 MiB of Flash.

Type	Location	Description	Compatible
CPU	on-chip	Espressif Xtensa LX7 CPU2	`espressif,xtensa-lx7`
ADC	on-chip	ESP32 ADC2	`espressif,esp32-adc`
Bluetooth	on-chip	Bluetooth HCI for Espressif ESP321	`espressif,esp32-bt-hci`
CAN	on-chip	ESP32 Two-Wire Automotive Interface (TWAI)1	`espressif,esp32-twai`
Clock control	on-chip	ESP32 Clock (Power & Clock Controller Module) Module1	`espressif,esp32-clock`
Counter	on-chip	ESP32 Counter Driver based on RTC Main Timer1	`espressif,esp32-rtc-timer`

Type	Location	Description	Compatible
	on-chip	ESP32 general-purpose timers13	`espressif,esp32-timer`
	on-chip	ESP32 counters4	`espressif,esp32-counter`
Cryptographic accelerator	on-chip	Espressif ESP32 SHA Hardware Accelerator1	`espressif,esp32-sha`
	on-chip	Espressif ESP32 family AES Hardware Accelerator1	`espressif,esp32-aes`
DMA	on-chip	ESP32 GDMA (General Direct Memory Access)1	`espressif,esp32-gdma`
Flash controller	on-chip	ESP32 flash controller1	`espressif,esp32-flash-controller`
GPIO & Headers	on-chip	ESP32 GPIO controller2	`espressif,esp32-gpio`
I2C	on-chip	ESP32 I2C2	`espressif,esp32-i2c`
I2S	on-chip	ESP32 I2S2	`espressif,esp32-i2s`
IIO	on-board	Voltage Divider1	`voltage-divider`
Input	on-chip	ESP32 touch sensor input1	`espressif,esp32-touch`

Type	Location	Description	Compatible
	on-board	Group of GPIO-bound input keys1	`gpio-keys`
Interrupt controller	**on-chip**	ESP32 Interrupt controller1	`espressif,esp32-intc`
IPM	**on-chip**	ESP32 soft inter processor message1	`espressif,esp32-ipm`
LED	**on-board**	Group of PWM-controlled LEDs1	`pwm-leds`
	on-board	Group of GPIO-controlled LEDs1	`gpio-leds`
Mailbox	**on-chip**	ESP32 soft mailbox1	`espressif,mbox-esp32`
Memory controller	**on-chip**	ESP32 pseudo-static RAM controller1	`espressif,esp32-psram`
MIPI-DBI	**on-chip**	ESP32 LCD-CAM display interface1	`espressif,esp32-lcd-cam-mipi-dbi`
Miscellaneous	**on-chip**	ESP32 LCD-CAM controller1	`espressif,esp32-lcd-cam`
	on-board	PWM passive buzzer1	`auxspaceev,pwm-buzzer`
MTD	**on-chip**	Flash node1	`soc-nv-flash`
Pin control	**on-chip**	ESP32 pin controller1	`espressif,esp32-pinctrl`

Type	Location	Description	Compatible
PWM	on-chip	ESP32 LED Control (LEDC)1	`espressif,esp32-ledc`
	on-chip	ESP32 Motor Control Pulse Width Modulator (MCPWM)2	`espressif,esp32-mcpwm`
Pyrotechnic Ignition	on-board	Basic pyro module1	`auxspaceev,basic-pyro`
RNG	on-chip	ESP32 TRNG (True Random Number Generator)1	`espressif,esp32-trng`
SDHC	on-chip	ESP32 SDHC controller1	`espressif,esp32-sdhc`
	on-chip	ESP32 SDHC controller slot2	`espressif,esp32-sdhc-slot`
Sensors	on-board	STMicroelectronics LSM6DSO32 6-axis IMU (Inertial Measurement Unit) sensor accessed through I2C bus1	`st,lsm6dso32`
	on-board	The BMP581 is a Barometric pressure sensor on I2C1	`bosch,bmp581`
	on-chip	ESP32 internal temperature sensor1	`espressif,esp32-temp`
	on-chip	ESP32 Pulse Counter (PCNT)1	`espressif,esp32-pcnt`
	on-chip	ESP32 UART3	

Type	Location	Description	Compatible
Serial controller			`espressif,esp32-uart`
	on-chip	ESP32 UART1	`espressif,esp32-usb-serial`
SPI	**on-chip**	ESP32 SPI controller2	`espressif,esp32-spi`
SRAM	**on-chip**	Generic on-chip SRAM2	`mmio-sram`
USB	**on-chip**	Espressif USB-OTG full-speed controller (DWC2 compatible) with the internal FS/LS PHY1	`espressif,esp32-usb-otg-fs`
Video	**on-chip**	ESP32 LCD-CAM camera interface1	`espressif,esp32-lcd-cam-dvp`
Watchdog	**on-chip**	ESP32 XT Watchdog Timer1	`espressif,esp32-xt-wdt`
	on-chip	ESP32 watchdog2	`espressif,esp32-watchdog`
Wi-Fi	**on-chip**	ESP32 SoC Wi-Fi1	`espressif,esp32-wifi`

Connections and IOs

- UART for programming and debugging
- SW1 for inputs
- Two seperated pin-pairs for the pyros

- Pins for battery
- 6-pin connector for expansion PCBs; don't throw heavy loads an the 3V3 pin, it may overheat the LDO.
- `rev2`: USB-C

Programming and Debugging

The `micrometer` board supports the runners and associated west commands listed below.

	flash	debug	attach	rtt	debugserver
esp32	(default)				
openocd		(default)			

> **⚠ Warning**
>
> On revision 1, if no µSD-Card is inserted the board will have trouble booting, since the SPI-SDHC driver has no way to detect card presence without a card-detect-pin!

Building

µMeter is built with `sysbuild` since it uses the MCUBoot boot loader:

```
# Revision 1 (default)
west build -p -b micrometer/esp32s3/procpu --sysbuild sensor_board

# Revision 2
west build -p -b micrometer@2/esp32s3/procpu --sysbuild sensor_board
```

Flashing

`west flash` is unfortunately not working with this setup, since the ESP requires the download mode to be set when booting and west flash is chain loading both images after one another.

Instead, flash both images in download mode and then just reset, using `esptool`:

```
esptool --chip esp32s3 -p /dev/tty<ESP_DEV> -b 921600 write-flash \
  0x0 build/mcuboot/zephyr/zephyr.bin \
  0x20000 build/sensor_board/zephyr/zephyr.signed.bin
```

Bench targets

Off-the-shelf development boards supported for bench testing of individual AURORA subsystems. Not intended for flight.

Raspberry Pi Pico (bench targets)

The Raspberry Pi Pico and Pico 2 families are supported as **bench / development targets**, not as flight boards. They are intended for testing the AURORA telemetry stack and other subsystems without populating a full sensor_board or micrometer PCB.

The same HC-12 wiring and the same AURORA Kconfig set work across all of the supported Pico variants. Pick the one whose silicon you want to validate against.

Supported variants

Upstream board target	SoC	Core	Notes
`rpi_pico/rp2040/w`	RP2040	Cortex-M0+	Same RP2040 silicon as the sensor_board v2 (Pico variant).
`rpi_pico2/rp2350a/m33/w`	RP2350A	Cortex-M33	RP2350A Arm core.
`rpi_pico2/rp2350a/hazard3/w`	RP2350A	Hazard3 (RISC-V)	Same RP2350A silicon, RISC-V core selected.

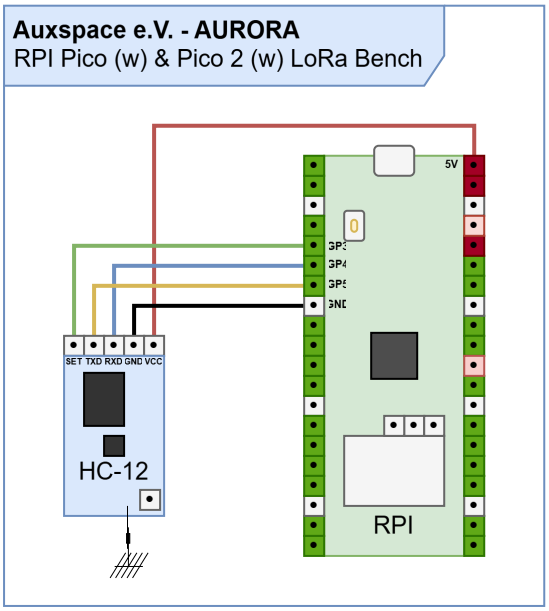
The non-W (no Wi-Fi) versions of each target also work. The bench overlay does not touch the Wi-Fi peripheral, but the `_w` variants are what we keep on the shelf.

What this target provides

The board overlays and `.conf` files AURORA layers on top of the upstream boards live next to the application.

They configure:

- **Console / shell** on USB CDC (`zephyr,console = &cdc_acm_uart`).
- **HC-12 telemetry radio** on UART1: TX=GP4, RX=GP5, SET=GP3 (active-low, used by the `telemetry hc12` shell to enter AT mode).
- The AURORA telemetry stack (`CONFIG_AURORA_TELEMETRY=y`) with the HC-12 shell command (`CONFIG_AURORA_TELEMETRY_HC12_SHELL=y`).
- Floating-point support for `printk`/log formatting (`CONFIG_CBPRINTF_FP_SUPPORT=y`), since telemetry payloads carry doubles.



No real IMU, barometer, or SD card is wired up. The bench target is meant to be paired with the `sim` snippet (fake sensors driving the state machine from a scripted profile) and the `nodisk` snippet (no flight data logger).

Note

Zephyr resolves board overlays and confs by exact board id, so each buildable target needs its own file pair. There is no implicit fall-back from `rpi_pico2/rp2350a/m33/w` to a generic `rpi_pico2` overlay. The Kconfig bodies are kept identical across the bench variants on purpose; the only per-target differences are the pinctrl header include and the pinmux group in the overlay.

Build

Pick the line that matches your hardware:

```
# Pico W (RP2040, Cortex-M0+)
west build -p -b rpi_pico/rp2040/w -S sim -S nodisk sensor_board

# Pico 2 W (RP2350A, Cortex-M33)
west build -p -b rpi_pico2/rp2350a/m33/w -S sim -S nodisk
sensor_board

# Pico 2 W (RP2350A, Hazard3 RISC-V)
west build -p -b rpi_pico2/rp2350a/hazard3/w -S sim -S nodisk
sensor_board
```

What each piece does:

- `-p`: pristine build; discard any previous build directory so Kconfig values from another target cannot bleed in.
- `-b <board>`: the upstream Zephyr board target.
- `-S sim`: applies the `sim` snippet, swapping real sensor drivers for the `fake_sensors` backend.
- `-S nodisk`: applies the `nodisk` snippet, disabling the flight data logger and its SD card requirement.
- `sensor_board`: the AURORA application under aurora/sensor_board/ being built.

Build output: `build/zephyr/zephyr.uf2`.

Flashing

Hold the BOOTSEL button on the board while plugging it in over USB, then drag-and-drop `build/zephyr/zephyr.uf2` onto the mass-storage device that appears. The board will reboot into the freshly flashed firmware.

Hardware wiring

Pico pin	Peripheral	Notes
GP3	HC-12 SET	Active-low; lets the shell switch the module into AT mode
GP4	HC-12 RXD	UART1 TX from the Pico

Pico pin	Peripheral	Notes
GP5	HC-12 TXD	UART1 RX into the Pico
3V3	HC-12 VCC	The HC-12 is 3.3 V tolerant
GND	HC-12 GND	

The console and `telemetry hc12` shell are reached over the Pico's USB CDC interface. No extra USB-to-serial adapter is needed.

Related

- The matching ground-station receiver: rec_zephyr.py.

Flight Logs

Recorded and simulated flight data lives in `aurora/flight_logs`. Each directory contains the raw InfluxDB line protocol export (`flights.influx`), a `state_audit` dump and rendered plots produced by `tools/plot_flight_data.py` (see plot_flight_data.py).

> **Note**
>
> Log payloads (`*.influx`, `*.png`, `state_audit`) are tracked via Git LFS. The documentation build requires `git lfs pull` (or `actions/checkout` with `lfs: true`) so that image references in the flight Readme files resolve to real files rather than LFS pointer blobs.

MULTIMETER Flight Logs

Source: `flight_logs/multimeter`

The M.E.T.A. rocket has its purpose in testing the avionics system for the MULTIMETER project. Here are the test flights of M.E.T.A. archived and open for everyone to see:

M.E.T.A. Test Flight 2026-05-03

Info

- AURORA Version: `0.3.1`
- Board: `esp32s3-micrometer/esp32s3/procpu`

Compile command

```
# Flight 1
python3 tools/plot_flight_data.py --flight flight_logs/multimeter/
2026-05-03/flight1 \
  --title "M.E.T.A. Flight 1 - 2026-05-03"

# Flight 2
python3 tools/plot_flight_data.py --flight flight_logs/multimeter/
2026-05-03/flight2 \
  --title "M.E.T.A. Flight 2 - 2026-05-03"
```

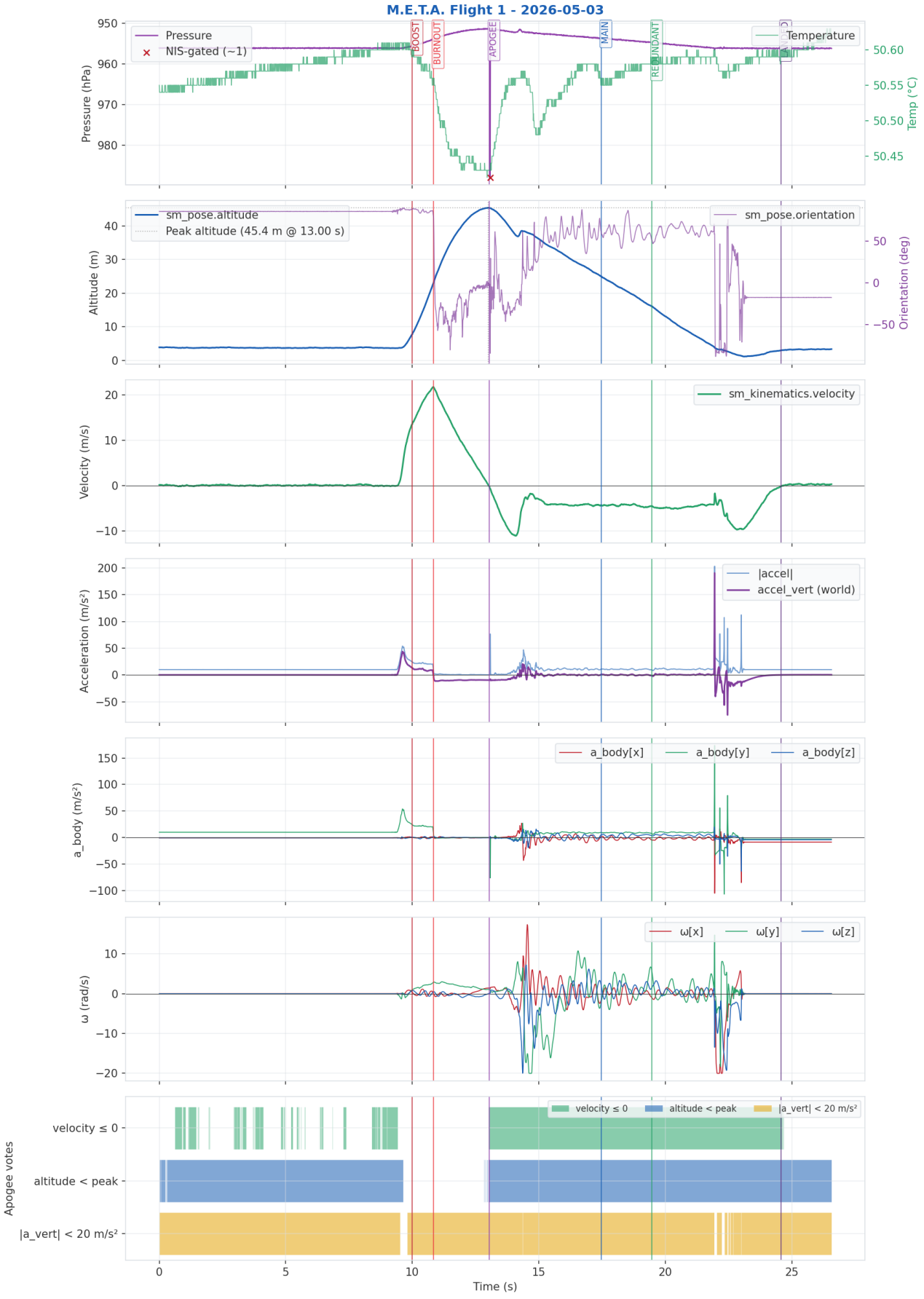
Flight Protocol

Two flights are recorded in `flight1` and `flight2`:

- Flight 1
  - parachute opened as expected
  - state machine cycled as expected
- Flight 2
  - parachute failed to open
  - state machine didn't go past `ARMED`
  - IMU Data is missing after ~500 seconds

Results

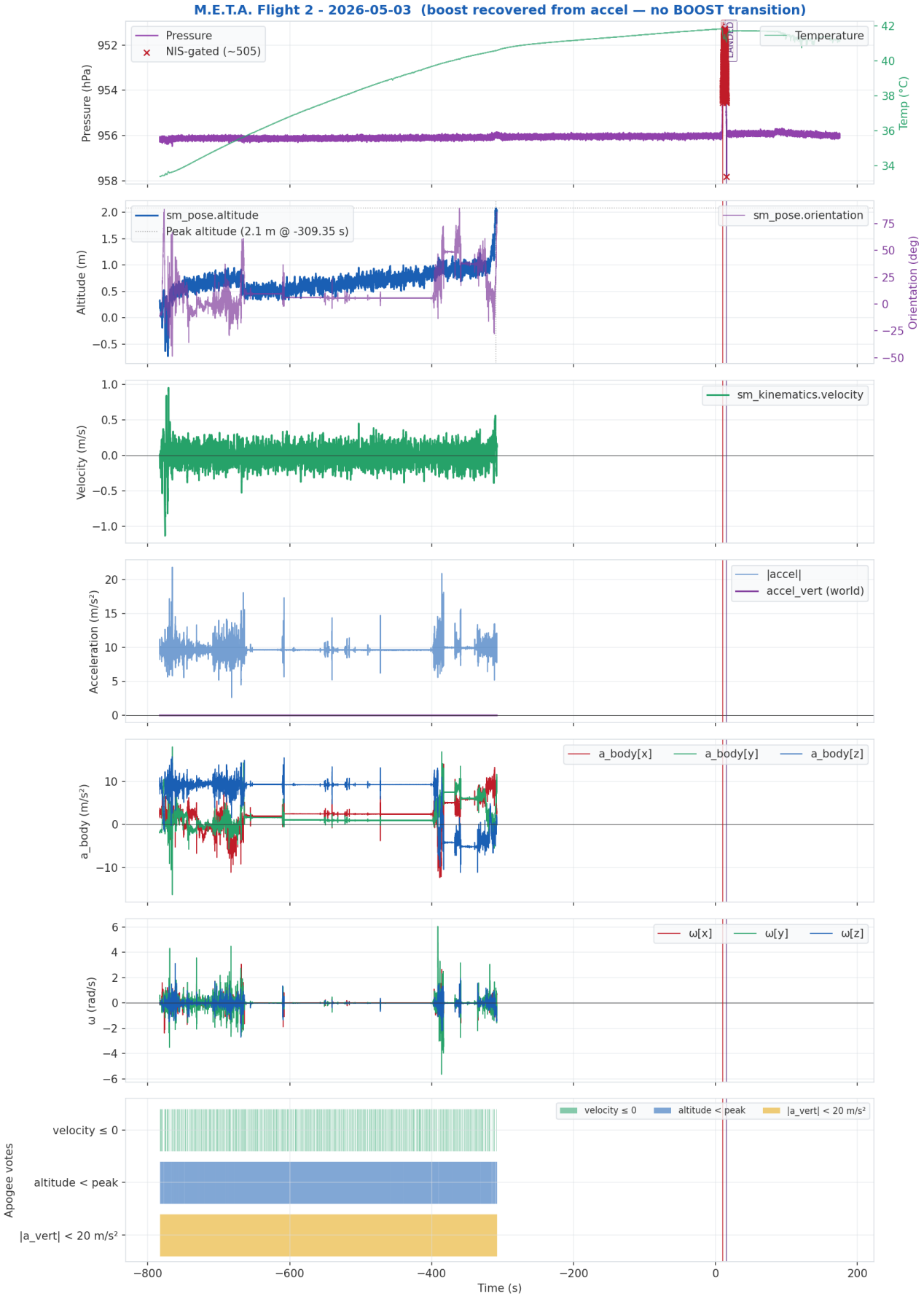
Flight1


M.E.T.A. Flight 1 - 2026-05-03
Pressure
NIS-gated (~1)
Temperature
BOOST
BURNOUT
APOGEE
MAIN
REDUNDANT
Pressure (hPa)
Temp (°C)
sm_pose.altitude
Peak altitude (45.4 m @ 13.00 s)
sm_pose.orientation
Altitude (m)
Orientation (deg)
sm_kinematics.velocity
Velocity (m/s)
|accel|
accel_vert (world)
Acceleration (m/s²)
a_body[x]
a_body[y]
a_body[z]
a_body (m/s²)
ω[x]
ω[y]
ω[z]
ω (rad/s)
velocity ≤ 0
altitude < peak
|a_vert| < 20 m/s²
Apogee votes
Time (s)

Flight2

⚠ Warning

This flight had an error, where the rocket launched before attitude calibration succeeded. This results in the state machine not detecting the `BOOST` phase. There is currently no explanation on why the IMU has stopped producing data.


M.E.T.A. Flight 2 - 2026-05-03 (boost recovered from accel — no BOOST transition)
Pressure
NIS-gated (~505)
Temperature
LANDED
Pressure (hPa)
Temp (°C)
sm_pose.altitude
Peak altitude (2.1 m @ -309.35 s)
sm_pose.orientation
Altitude (m)
Orientation (deg)
sm_kinematics.velocity
Velocity (m/s)
|accel|
accel_vert (world)
Acceleration (m/s²)
a_body[x]
a_body[y]
a_body[z]
a_body (m/s²)
ω[x]
ω[y]
ω[z]
ω (rad/s)
velocity ≤ 0
altitude < peak
|a_vert| < 20 m/s²
Apogee votes
Time (s)

M.E.T.A. Test Flight 2026-04-26

Info

- AURORA Version: `0.3.0`
- Board: `esp32s3-micrometer/esp32s3/procpu`

Compile command

```
# Flight 1
python3 tools/plot_flight_data.py --flight flight_logs/2026-04-26/
flight1 \
  --pre-boost 4 \
  --title "AURORA Flight 1 - 2026-04-26"

# Flight 2
python3 tools/plot_flight_data.py --flight flight_logs/2026-04-26/
flight2 \
  --pre-boost 4 \
  --title "AURORA Flight 2 - 2026-04-26"
```

Flight Protocol

Two flights are recorded in `flight1` and `flight2`:

- Flight 1
  - parachute opened as expected
  - state machine cycled as expected
- Flight 2
  - parachute opened as expected
  - state machine cycled as expected

Note

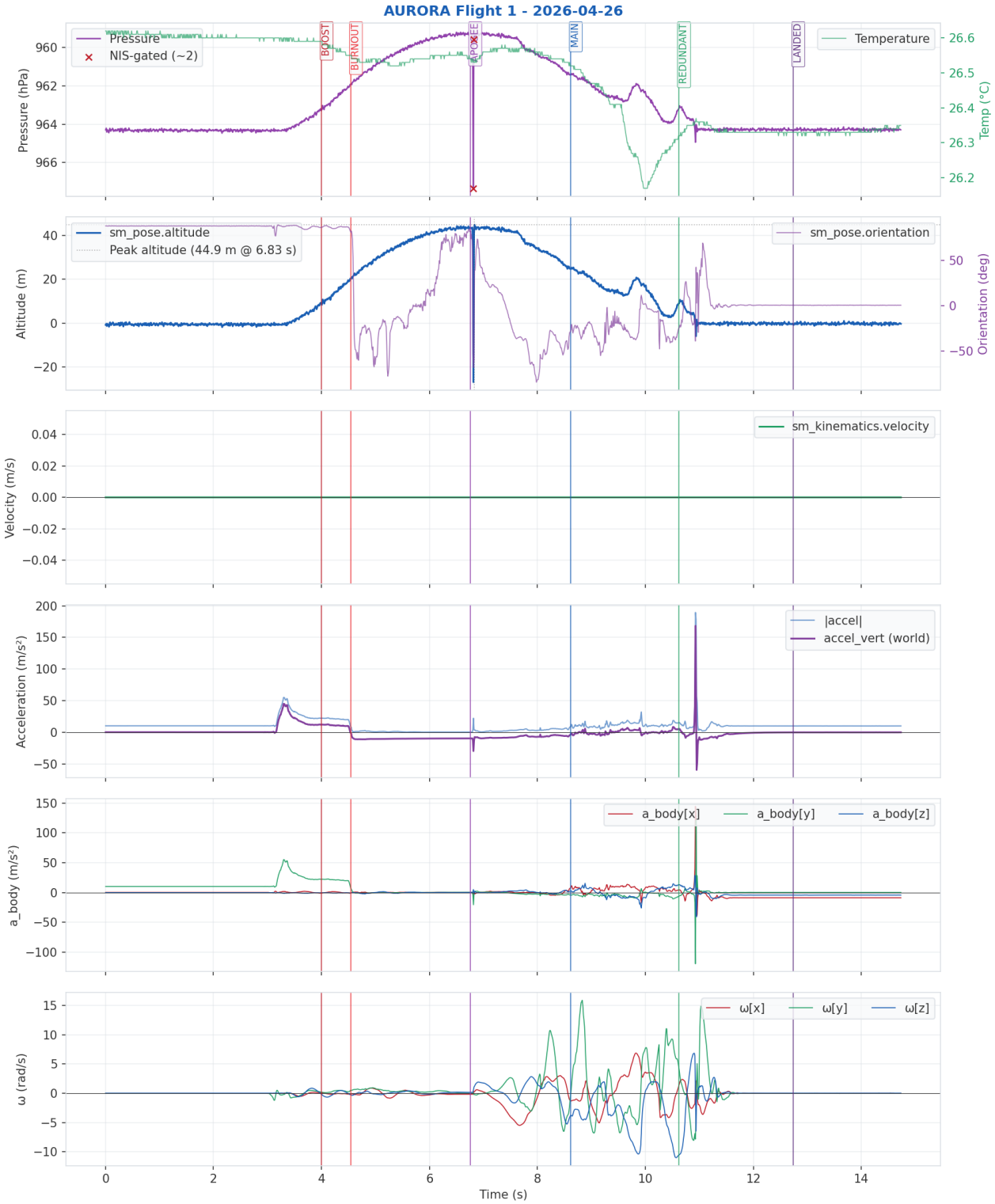
The velocity graph is a flat line, not because the filter didn't work, but because the logger had an issue with a pointer variable that wasn't accessed correctly. This issue has been fixed in `f788cd50cffa705b049117e93fb963d218b2784d`

Note

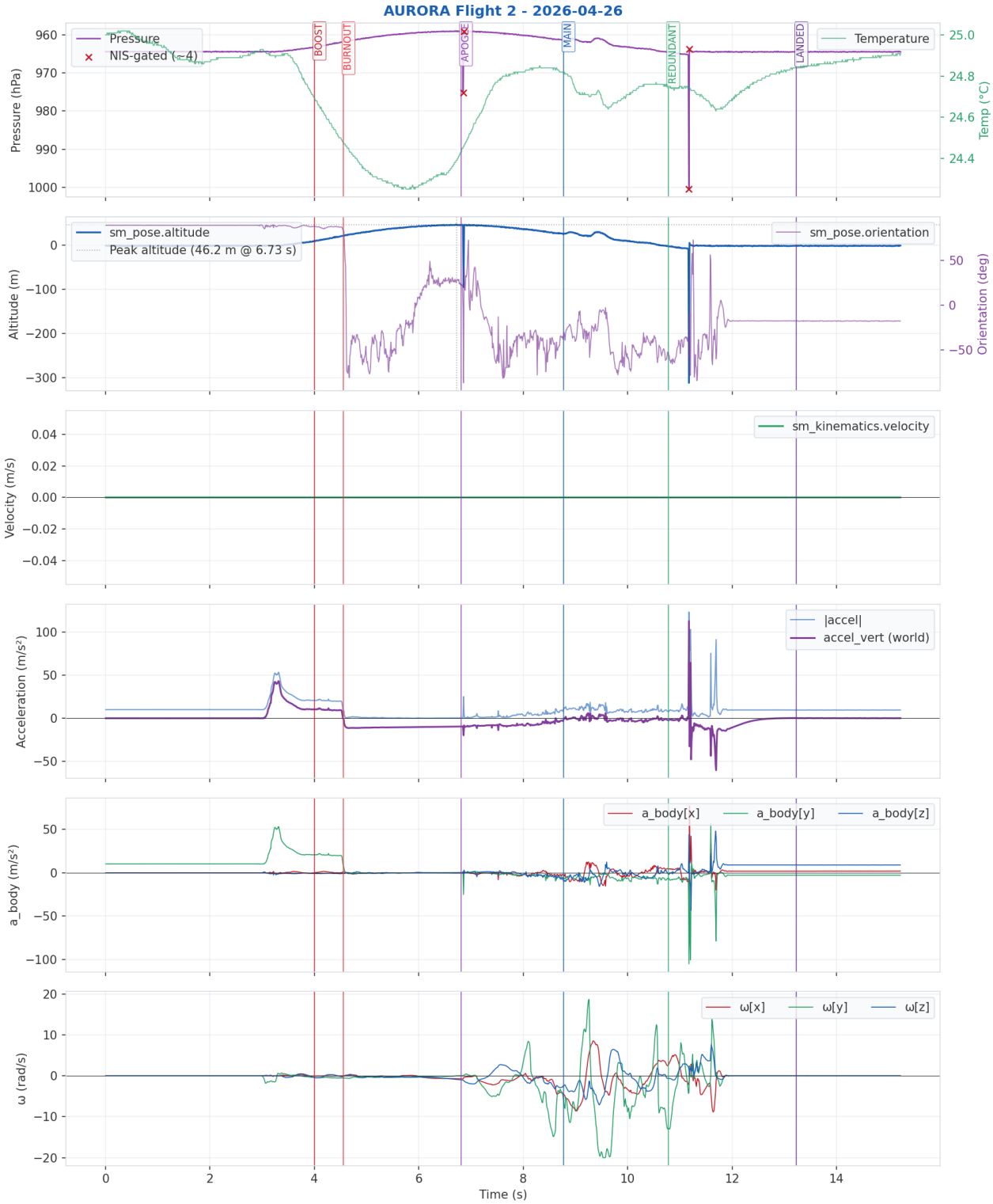
Voting was also disabled in the graphs, since the faulty data would have shown the votes as always triggered.

Results

Flight1



Flight2


AURORA Flight 2 - 2026-04-26
Pressure
NIS-gated (≈4)
Temperature
BOOST
BURNOUT
APOGEE
MAIN
REDUNDANT
LANDED
Pressure (hPa)
Temp (°C)
sm_pose.altitude
Peak altitude (46.2 m @ 6.73 s)
sm_pose.orientation
Altitude (m)
Orientation (deg)
sm_kinematics.velocity
Velocity (m/s)
|accel|
accel_vert (world)
Acceleration (m/s²)
a_body[x]
a_body[y]
a_body[z]
a_body (m/s²)
ω[x]
ω[y]
ω[z]
ω (rad/s)
Time (s)

M.E.T.A. Test Flight 2026-04-18

Info

- AURORA Version: `0.2.1`
- Board: `esp32s3-micrometer/esp32s3/procpu`

Compile command

```
python3 tools/plot_flight_data.py --flight flight_logs/2026-04-18 \
  --r-meas 6.0 \
  --pre-boost 4 \
  --post-end 4
```

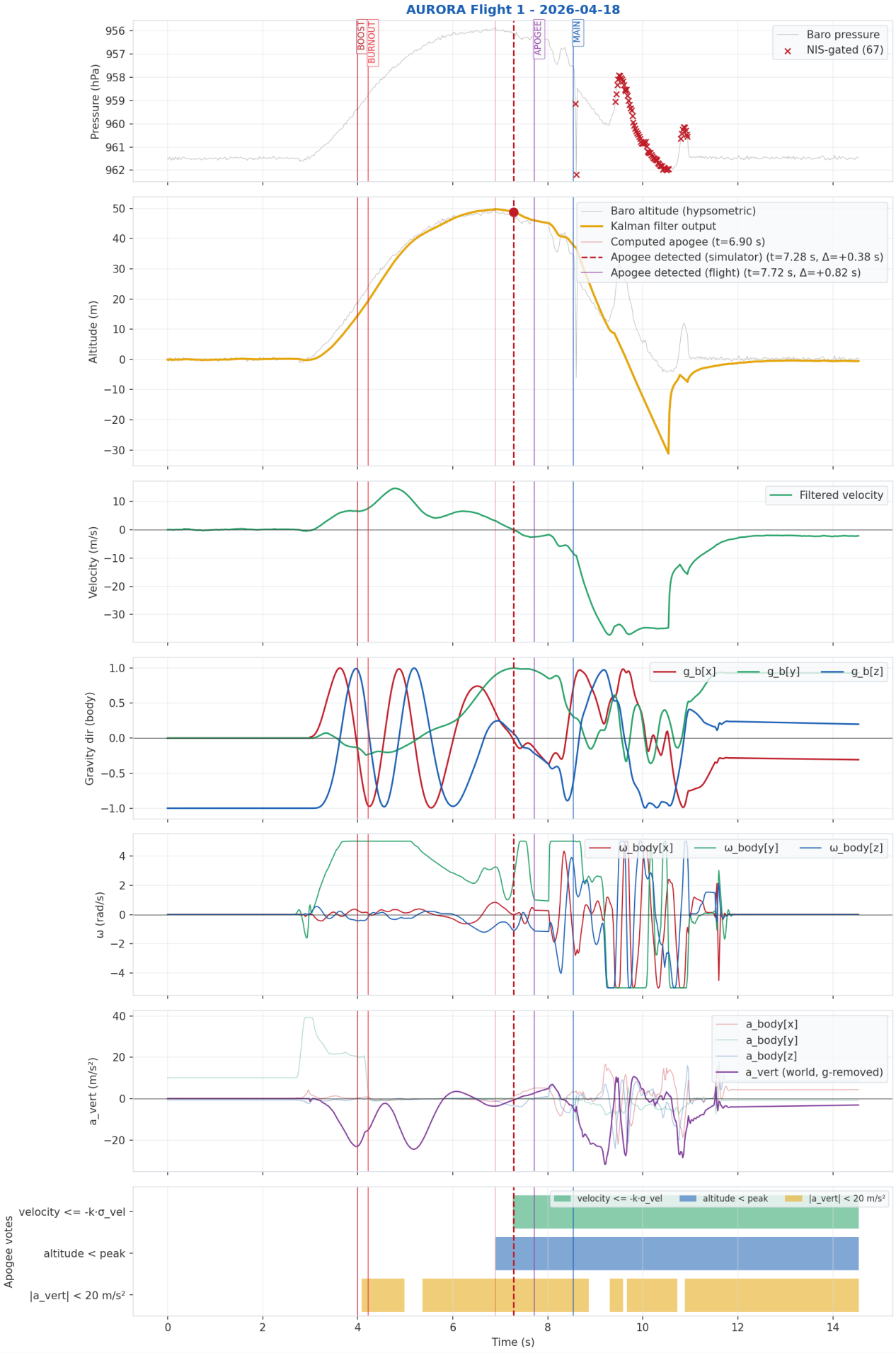
Flight Protocol

Two flights are recorded in `flights.influx`:

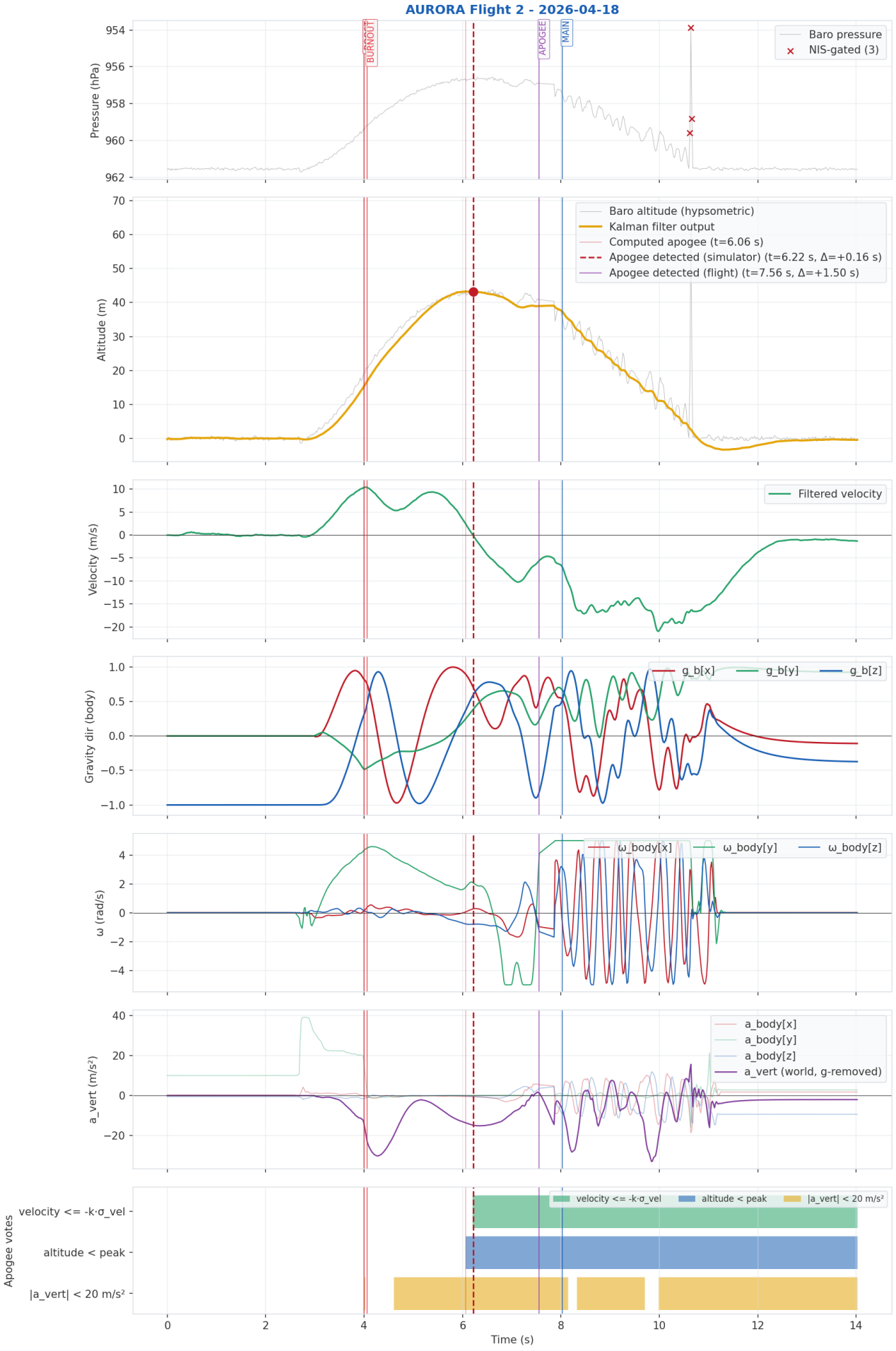
- Flight1:
  - parachute opened too late
  - state machine didn't go from MAIN to REDUNDAND
- Flight2:
  - parachute didn't open; pyro fired on the ground
  - state machine didn't go from MAIN to REDUNDAND

Results

Flight 1


AURORA Flight 1 - 2026-04-18
BOOST
BURNOUT
APOGEE
MAIN
Pressure (hPa)
Baro pressure
NIS-gated (67)
Altitude (m)
Baro altitude (hypsometric)
Kalman filter output
Computed apogee (t=6.90 s)
Apogee detected (simulator) (t=7.28 s, Δ=+0.38 s)
Apogee detected (flight) (t=7.72 s, Δ=+0.82 s)
Velocity (m/s)
Filtered velocity
Gravity dir (body)
g_b[x]
g_b[y]
g_b[z]
ω (rad/s)
ω_body[x]
ω_body[y]
ω_body[z]
a_vert (m/s²)
a_body[x]
a_body[y]
a_body[z]
a_vert (world, g-removed)
Apogee votes
velocity <= -k·σ_vel
altitude < peak
|a_vert| < 20 m/s²
Time (s)

Flight 2


AURORA Flight 2 - 2026-04-18
BURNOUT
APOGEE
MAIN
Baro pressure
NIS-gated (3)
Pressure (hPa)
Baro altitude (hypsometric)
Kalman filter output
Computed apogee (t=6.06 s)
Apogee detected (simulator) (t=6.22 s, Δ=+0.16 s)
Apogee detected (flight) (t=7.56 s, Δ=+1.50 s)
Altitude (m)
Filtered velocity
Velocity (m/s)
g_b[x]
g_b[y]
g_b[z]
Gravity dir (body)
ω_body[x]
ω_body[y]
ω_body[z]
ω (rad/s)
a_body[x]
a_body[y]
a_body[z]
a_vert (world, g-removed)
a_vert (m/s²)
velocity <= -k·σ_vel
altitude < peak
|a_vert| < 20 m/s²
Apogee votes
Time (s)

Flight Simulation

Source: `flight_logs/sim_fetch`

Info

- AURORA Version: `0.2.1`
- Board: `native_sim`

Compile command

```
# Building and running the simulator first:
west build -p -b native_sim/native sensor_board/ -- \
  -DCONFIG_AURORA_FAKE_SENSORS_REPLAY=y

python3 ./tools/sim_fetch.py pexpect \
  build/zephyr/zephyr.exe \
  --await-ready "Attitude calibrated" \
  --pre-command "sim launch" \
  --wait-for "converted flight_log" \
  --wait-timeout 600 \
  --fetch /RAM:/data/flight_0.influx flights.influx \
  --fetch /RAM:/state/audit.0 state_audit \
  --fetch /RAM:/data/flight_0.csv flights.csv \
  -v
```

```
python3 ./tools/plot_flight_data.py \
  --flight flight_logs/sim_fetch \
  --title "Simulated Flight (v0.4.1)"
```

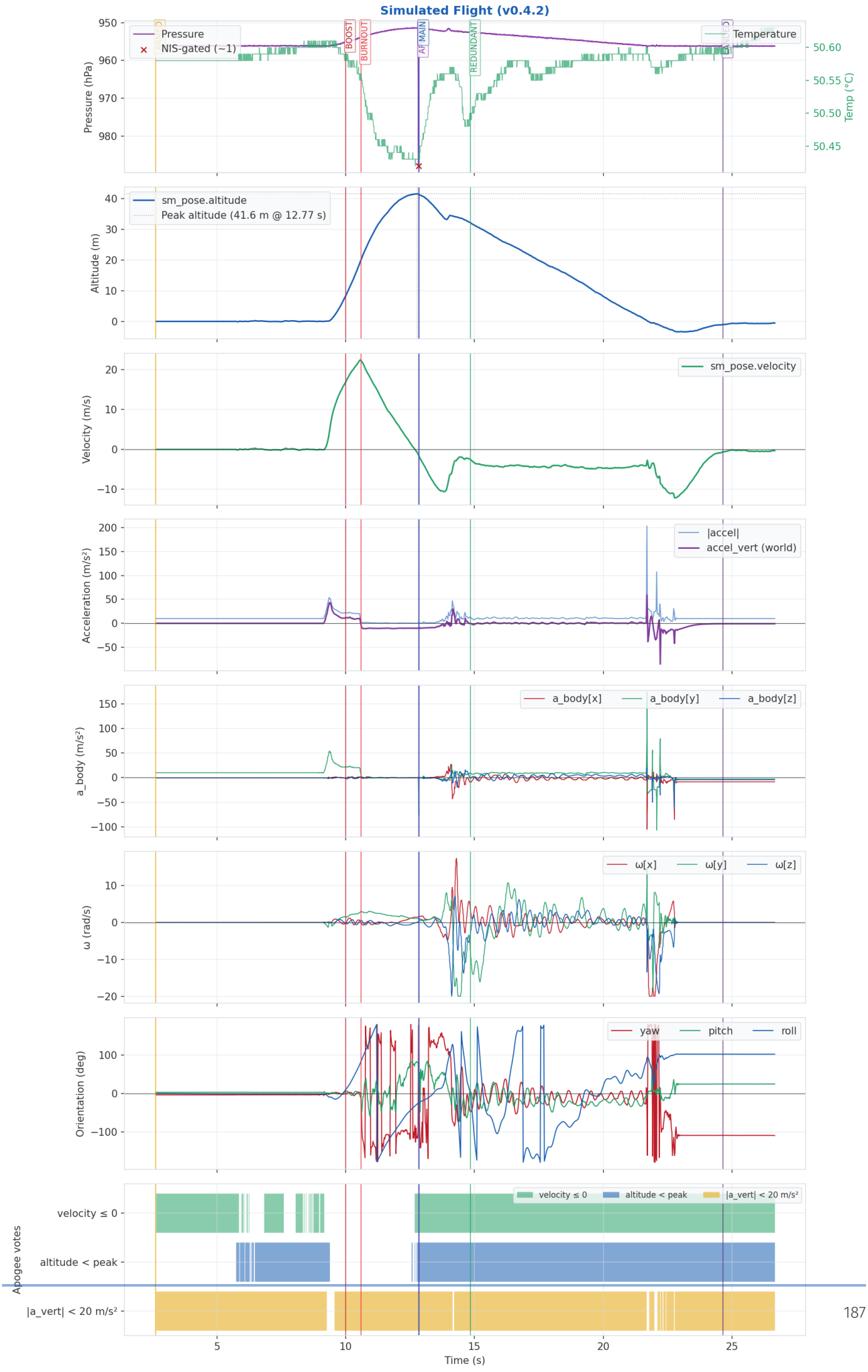
Flight Protocol

One flight is recorded in `flights.influx`:

- parachute opened fine
- state machine behaved as expected

Results

Flight 1


Simulated Flight (v0.4.2)
Pressure
NIS-gated (~1)
Temperature
BOOST
BURNOUT
AP MAIN
REDUNDANT
LANDED
Pressure (hPa)
Temp (°C)
sm_pose.altitude
Peak altitude (41.6 m @ 12.77 s)
Altitude (m)
sm_pose.velocity
Velocity (m/s)
|accel|
accel_vert (world)
Acceleration (m/s²)
a_body[x]
a_body[y]
a_body[z]
a_body (m/s²)
ω[x]
ω[y]
ω[z]
ω (rad/s)
yaw
pitch
roll
Orientation (deg)
velocity ≤ 0
altitude < peak
|a_vert| < 20 m/s²
Apogee votes
Time (s)

Indices and tables

- Index
- Search Page

Created using Sphinx 9.1.0.