HuskySat-2 Software Design Document

Table of Contents

1 Introduction and System Context **5**

2 Architecture Overview **6**
2.1 Software Topology . . . 7

3 F' Context **8**
3.1 Framework Structure . . . 8
3.2 Core Concepts . . . 8
3.3 From Component SDD to FPP and C++ . . . 9
3.4 Topology Assembly . . . 9
3.5 Pre-Built Services and Subtopologies . . . 10
3.6 Design Patterns Used . . . 10
3.6.1 App-Manager-Driver . . . 10
3.6.2 Rate Groups . . . 10
3.6.3 Health Checking . . . 10
3.6.4 Manager-Worker . . . 10
3.6.5 Subtopologies and Common Ports . . . 10
3.7 Ground Interface . . . 11

I Level 4 - System Services and Control Layer **12**

4 SatStateMachine **13**

5 Svc::CmdDispatcher **19**

6 F' System Services **20**
6.1 Svc::EventManager . . . 20
6.2 Svc::TlmChan and Svc::TlmPacketizer . . . 20
6.3 Svc::Health . . . 20
6.4 Svc::RateGroupDriver and Svc::ActiveRateGroup . . . 20
6.4.1 HuskySat-2 rate-group offsets . . . 21
6.5 Svc::Version . . . 21
6.6 Svc::AssertFatalAdapter . . . 21

7 Svc::FatalHandler **22**

II Level 3 - Application Layer **24**

8 F' Application and Data Services **25**
8.1 Svc::ComQueue . . . 25
8.2 Svc::FrameAccumulator . . . 25
8.3 Svc::FprimeRouter . . . 25
8.4 Svc::FprimeFramer and Svc::FprimeDeframer . . . 25
8.5 Svc::Ccsds::TmFramer and Svc::Ccsds::TcDeframer . . . 26
8.6 Svc::Ccsds::SpacePacketFramer and Svc::Ccsds::SpacePacketDeframer . . . 26
8.7 Svc::Ccsds::ApidManager . . . 26
8.8 Svc::ComAggregator . . . 26
8.9 Svc::FileUplink and Svc::FileDownlink . . . 26
8.10 Svc::FileManager . . . 27

8.11 Svc::PrmDb . . . 27
8.12 Svc::DpManager, Svc::DpWriter, and Svc::DpCatalog . . . 27

9 AdcsApplication **28**

10 DataCollectionApplication **34**

11 ComApplication **38**

12 EPSApplication **47**

13 ScienceApplication **51**

14 ThermalApplication **55**

III Level 2 - Hardware Managers 59

15 ADCS Hardware Managers **60**
15.1 ImmuManager . . . 61
15.2 SunSensorManager . . . 64

16 StarTrackerManager **68**
16.1 MagnetorquerManager . . . 72

17 Communications Hardware Managers **75**
17.1 TmtcRadioManager . . . 76

18 Payload Hardware Managers **79**

19 CameraManager **80**
19.1 GnssManager . . . 82

20 Electrical Power System Hardware Managers **85**
20.1 CurrentSensorManager . . . 86
20.2 DeployPanelsManager . . . 89
20.3 MpptManager . . . 91
20.4 WatchdogPinger . . . 94

21 Thermal Hardware Managers **95**
21.1 HeaterManager . . . 96
21.2 TemperatureSensorManager . . . 100

IV Level 1 - Drivers 103

22 Operating-System Abstraction Layer (OSAL) **104**
22.1 F' OSAL and Linux on the BeagleBone Black . . . 104
22.2 Os::File . . . 104
22.3 Os::Task . . . 104
22.4 Os::Queue . . . 104
22.5 Os::Mutex . . . 105
22.6 Os::Time . . . 105

23 LinuxPwmDriver **106**

24 Drv::LinuxUartDriver **108**

25 Drv::LinuxI2cDriver **109**

26 Drv::LinuxSpiDriver **110**

27 Drv::LinuxGpioDriver **111**

V Appendix 112

28 Hardware and Platform Inventory **113**

29 Radiation and Storage Architecture **114**
29.1 Historical Architecture . . . 114
29.2 Current Storage Architecture . . . 114
29.3 Storage Partitioning Policy . . . 115
29.4 Storage Recovery Policy . . . 116
29.5 Runtime RAM Contents . . . 116
29.6 Software-Level Radiation and Fault Mitigation . . . 116
29.7 Computational Load and Power Protection . . . 117
29.8 Data Retention and Ground Operations . . . 118

30 Verification and Test Plan **119**
30.1 Component-Level Tests . . . 119
30.2 GDS Deployment Tests . . . 119
30.3 Subsystem Tests . . . 119
30.4 Integrated Topology Tests . . . 119
30.5 Verification Evidence . . . 120
30.6 Related Verification Artifacts . . . 120

31 Software Development Process **121**
31.1 Configuration Management . . . 121
31.2 Component Design and Coding Standards . . . 121
31.3 Branching, Review, and Merge . . . 121
31.4 Test, Release, and Issue Tracking . . . 122
31.5 Roles and Accountability . . . 122

32 Revision History and Update Log **123**

33 Acronyms **125**

34 References **126**
34.1 F' Framework and FPP . . . 126
34.2 F' Drivers and OSAL . . . 126
34.3 F' Services and Components . . . 126
34.4 HuskySat-2 and Mission References . . . 127

1 Introduction and System Context

This document defines the HuskySat-2 flight-software design and the interfaces between its software layers. It is written for a 3U CubeSat flight computer running Linux and F' (F Prime). The document introduces the software context first, then describes the Level 4 system-services and control components, Level 3 applications, Level 2 hardware managers, and Level 1 drivers.

HuskySat-2 is a 3U CubeSat mission that uses optical-navigation experiments to validate the LOST, FOUND, and SCOPE algorithms. The flight software supports that objective by commanding experiments, collecting and processing payload data, retaining selected products for downlink, and keeping the spacecraft in a safe, recoverable operating state while it performs those activities.

The scope of this Software Design Document is the HuskySat-2 flight-software repository and the deployment architecture required to run it on the BeagleBone Black flight computer. It covers:

- software interfaces, ports, commands, events, telemetry, parameters, and state machines for components represented in this repository;
- Linux-backed Level 1 drivers and the Level 2 hardware managers that own device protocols and recovery behavior;
- Level 3 subsystem applications, ADCS algorithms, and the F' framework components used by the application topology;
- the intended decomposition of subsystem topologies and their cross-layer connections;
- integration of the LOST, FOUND, and SCOPE mission libraries; and
- the staged verification path from unit tests through GDS, subsystem, and integrated deployment testing.

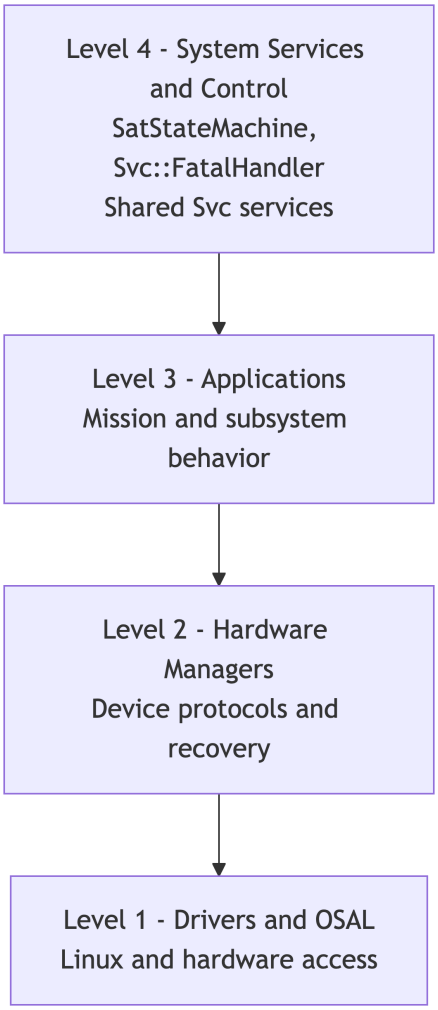
The Level 4 system-services and control layer contains the satellite state machine, deployment-wide F' services, and fatal handling; hardware-reset ownership remains with the application-manager interfaces described in that layer.

2 Architecture Overview

This page describes the HuskySat-2 software architecture only. It does not describe the spacecraft electrical architecture or the physical system. The software is organized as a set of abstraction layers: higher levels express more mission policy, while lower levels are closer to Linux and the hardware interface.

Level	Software responsibility
Level 4 - System Services and Control	Provides deployment-wide F' command, event, telemetry, health, timing, version, and fatal-response services; coordinates satellite-wide modes.
Level 3 - Applications	Implements subsystem behavior, mission operations, communications policy, data products, and algorithm selection.
Level 2 - Hardware Managers	Owns device protocols, initialization, device-specific state, telemetry, and recovery requests.
Level 1 - Drivers and OSAL	Exposes Linux buses, GPIO/PWM, serial devices, files, tasks, queues, synchronization, and time.

The abstraction relationship is intentionally top-down: Level 4 depends on application interfaces, applications depend on manager interfaces, and managers depend on drivers or F' OSAL services. A lower layer does not make mission decisions on behalf of a higher layer.



The level names describe software abstraction, not task priority or physical placement. F' services used by the applications are documented individually in the Level 3 section; they remain reusable framework components rather than mission-specific hardware managers.

2.1 Software Topology

The top-level software integration diagram below complements the abstraction-layer diagram above. It shows the major application boundaries, shared system services, fatal-event path, command and telemetry services, and the connections between software layers. Individual component pages define the detailed ports and behavior.

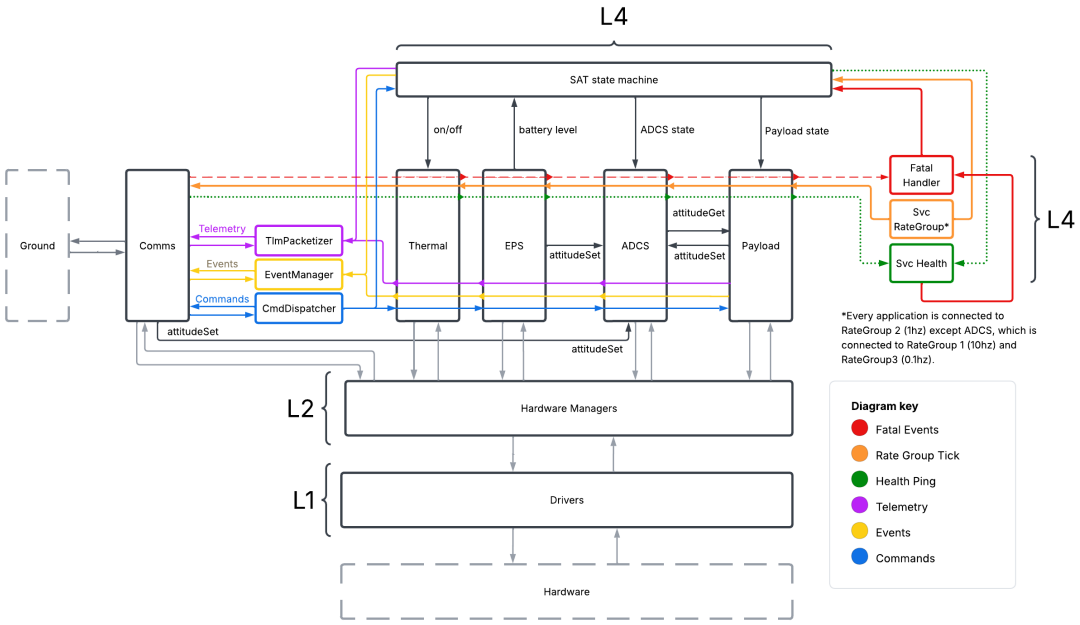


Figure 2.1: High-level software topology showing the satellite state machine, system services, subsystem applications, hardware managers, drivers, and hardware.

3 F' Context

F' (F Prime) is a component-driven embedded flight-software framework developed at NASA JPL for small-scale spacecraft and other embedded systems. HuskySat-2 uses F' to separate reusable framework infrastructure from mission-specific applications, hardware managers, and drivers.

3.1 Framework Structure

The F' repository is organized around several layers of reusable infrastructure:

Area	Role in a deployment
`Fw/`	Runtime types and core abstractions such as buffers, commands, logs, telemetry, time, and state machines.
`Svc/`	Reusable active, queued, and passive service components for CDH, timing, communications, files, data products, and diagnostics.
`Drv/`	Operating-system and bus drivers, including Linux UART, SPI, I2C, and GPIO components.
`Os/`	Operating-system abstraction layer used by framework and deployment code.
`Fpp/`	FPP modeling and code-generation toolchain for component and topology definitions.
`Ref/`	Reference deployment showing component instantiation, initialization, wiring, and startup.

3.2 Core Concepts

F' components communicate through typed ports. A topology creates component instances, initializes them, connects their ports, registers commands, loads parameters, and starts active components. FPP describes the interfaces and topology structure; C++ provides implementation logic and deployment-specific assembly.

F' supports four important port execution styles:

Port style	Execution behavior	Typical use
Output	Calls the connected input through the caller's thread.	Synchronous request or data delivery.
Synchronous input	Executes immediately in the calling thread and may return data.	Short, deterministic operations.
Asynchronous input	Queues work for an active component's execution context.	Commands and longer-running work.
Guarded input	Executes synchronously while protecting shared state with a mutex.	Thread-safe access to shared data.

The main F' data interfaces are commands, events, telemetry channels, and parameters. Commands represent uplinked operator instructions and are routed through `Svc::CmdDispatcher`. Events record component activity and faults and are routed through `Svc::EventManager`. Telemetry channels represent current state and are collected by `Svc::TlmChan` and `Svc::TlmPacketizer`. Parameters hold configurable values and may be persisted by `Svc::PrmDb`.

3.3 From Component SDD to FPP and C++

The component SDD is the design contract for implementation: its port, command, event, telemetry, parameter, and state-machine descriptions are translated into an FPP component definition. The FPP file is the interface source of truth, similar to an API or header, but it is also machine-readable so F' can check connections and generate code.

This translation is close to mechanical, and that's the point. Every table in the SDD maps to a specific FPP construct:

SDD element	FPP construct
Port table (name, direction, type, purpose)	`sync input port` / `output port` declarations
Commands table (mnemonic, args)	`async command` / `sync command` declarations with opcodes
Events (e.g. `WARNING_HI`, activity logs)	`event` declarations with severity and format strings
Telemetry channels	`telemetry` declarations with type and update rate
Parameters (thresholds, config values)	`param` declarations with default values
Operational behavior / flow diagrams	The body of the hand-written handler functions

So a port and command from an SDD become declarations in a component model:

```
active component Example {
    sync input port dataIn: ExampleData
    output port resultOut: ExampleResult
    async command PROCESS(value: U32)
    event ProcessFailed(reason: string) severity warning high format "Process failed: {}"
    telemetry ProcessCount: U32
    param PROCESS_THRESHOLD: U32 default 100
}
```

Running the F' tooling (often through `fprime-util new --component`, `fprime-util impl`, and the deployment build) generates typed C++ base files such as `ExampleComponentAc.hpp` and `ExampleComponentAc.cpp`. These generated files handle everything that is boilerplate once the interface is fixed: command opcode dispatch and argument deserialization, port call proxies and connection-time type checking, telemetry channel packing, event formatting and routing to the ground interface, and parameter loading from `PrmDb`. None of that is something a developer writes by hand or needs to design — it falls out automatically once the SDD's tables are captured in FPP.

What's left for the developer is exactly the part the SDD already spells out in its Operational Behavior section: the step-by-step flow for each handler (what to read, what to check, what to emit, what to cache). The hand-written `Example.hpp` and `Example.cpp` files fill in those generated handler stubs with that logic. So the SDD isn't a partial description that implementers have to fill gaps in from experience — it's a complete description split across two things the tooling already understands how to consume: the interface tables (which become FPP, checked by the compiler) and the behavior narrative (which becomes the handler bodies).

This is also why "Inspection" is a valid verification method for most SDD requirements: if a requirement is fully expressed as a port, command, or event in the SDD, verifying it means confirming the FPP declares that port/command/event correctly and the generated code compiles against it — the toolchain itself catches interface mismatches (wrong types, missing connections, unconnected ports) that would otherwise require a review to find.

See the F' component development process, Hello World component tutorial, and autocoded functions reference for the complete workflow.

3.4 Topology Assembly

The deployment topology is responsible for more than simply listing components. It must:

1. instantiate components with the required queues, priorities, and stack sizes;
2. call `init()` and apply deployment configuration;
3. connect typed ports and validate the intended data flow;

4. register commands and load parameters;
5. configure rate groups and health-monitoring entries; and
6. start active components in a safe order.

HuskySat-2 will preserve the same component interfaces when moving from a GDS deployment to the BeagleBone Black flight deployment. That keeps application and manager tests independent of the final physical bus wiring.

3.5 Pre-Built Services and Subtopologies

F' provides reusable components and importable subtopologies. A subtopology is a pre-wired group of component instances and connections, while its individual components still have their own behavior and interfaces. HuskySat-2 documents the individual F' components used by the Level 3 application layer so their role is visible without reproducing the entire upstream repository summary.

The principal upstream groupings are:

- `CdhCore`: command dispatch, events, health, version, text logging, and fatal adaptation;
- `ComCcsds`: packet routing, CCSDS framing/deframing, aggregation, and the byte-stream communications bridge;
- `FileHandling`: file uplink, file downlink, file management, and parameter storage; and
- `DataProducts`: data-product allocation, writing, cataloging, and downlink preparation.

3.6 Design Patterns Used

3.6.1 App-Manager-Driver

Applications own mission behavior, managers translate device-specific protocols and recovery behavior, and drivers expose hardware or operating-system access. This prevents mission applications from depending directly on bus details and allows managers to be tested against simulated drivers.

3.6.2 Rate Groups

`Svc::RateGroupDriver` divides a primary clock into multiple rates, while `Svc::ActiveRateGroup` invokes configured components at those rates. Control loops, telemetry collection, and background work can therefore have separate scheduling requirements. Blocking work in a passive rate group can delay other work; active rate groups isolate execution at the cost of thread scheduling and possible jitter.

3.6.3 Health Checking

`Svc::Health` periodically pings critical active components through their ping ports. Components echo the ping key promptly; missed responses become warning or fatal health events according to configured thresholds. Background workers and components without mission-critical liveness obligations may be excluded.

3.6.4 Manager-Worker

The manager-worker pattern separates high-priority command handling from long-running background work. The manager tracks state and cancellation while a worker performs the operation and reports completion. This pattern is relevant to file operations and future workloads that must not block command responsiveness.

3.6.5 Subtopologies and Common Ports

Subtopologies package reusable component instances and connections for import into a deployment. Common F' port patterns include synchronous gets, asynchronous callbacks, parallel port arrays, and synchronous cancellation. These patterns make interfaces explicit and allow FPP to validate connection compatibility before deployment.

3.7 Ground Interface

The F' ground interface is two-sided and layered. The uplink path follows the general form:

`Driver -> FrameAccumulator -> Deframer -> FprimeRouter -> CmdDispatcher`

The downlink path follows:

`Svc::EventManager/Svc::TlmChan -> Svc::ComQueue -> Framer -> Driver`

HuskySat-2's CCSDS service components fill in the framing and packet-routing stages around `TmtcRadioManager` and `LinuxUartDriver`.

The command and telemetry list, including both F' events and telemetry channels, can be viewed in the CTL document. The CTL document is maintained separately and is not part of this SDD.

Refer to the separate Ground Data System (GDS) document for the ground-station hardware, ground software, networking, and operator-facing workflows. This SDD defines the flight-software boundary: command decoding, validation, dispatch, status response, telemetry, and event interfaces that the ground system consumes. It does not duplicate the ground-software design; the GDS document is the authoritative source for that material.

Part I

Level 4 - System Services and Control Layer

4 SatStateMachine

Overview

`SatStateMachine` is the Layer 4 "Mission Orchestration" component. It is an **Active** component, instantiated at the **top-level topology** since it is not scoped to any single subtopology. It evaluates the satellite's top level operating mode among four sibling states, Safe, Downlink, Science, and Charge, reevaluated every 1 Hz tick. All condition inputs arrive precomputed via typed ports from Layer 3 applications.

`SatStateMachine` owns the mode to mode translation table On every mode transition it is intended to command each Layer 3 application component's own operating mode via a dedicated typed port.

Requirements

ID	Requirement	Verification
HS2-SAT-001	SatStateMachine shall boot into Safe mode on deploy/reboot	Inspection
HS2-SAT-002	SatStateMachine shall enter Safe mode from any state on ground command `SAFE_MODE` if `checkoutComplete` is not set, else enter Safe mode	Inspection
HS2-SAT-003	SatStateMachine shall exit Safe to Charge only on ground command `SAFE_EXIT`, and only after the `CHECKOUT_COMPLETE` ground command has been received at least once; otherwise the command is denied and SatStateMachine remains in Safe	Inspection
HS2-SAT-004	SatStateMachine shall evaluate which of Downlink, Science, or Charge to enter every 1 Hz tick while not in Safe mode, in strict priority order: Downlink, then Science, then Charge (fallback)	Inspection
HS2-SAT-005	SatStateMachine shall log an event on every mode entry and exit	Inspection
HS2-SAT-006	SatStateMachine shall report its current mode, checkout status, and cached condition inputs as telemetry each tick	Inspection
HS2-SAT-007	SatStateMachine shall command each Layer 3 application component's operating mode per its translation table	Inspection
HS2-SAT-008	SatStateMachine shall respond to `Svc.Health` pings	Inspection
HS2-SAT-009	SatStateMachine shall support a `RESET` command that forces reentry to `SAFE`.	Inspection
HS2-SAT-010	SatStateMachine shall not enter Downlink unless `ComApplication` reports downlink readiness via `commsReadyIn`	Inspection
HS2-SAT-011	SatStateMachine shall command `AdcsApplication` into `Detumble` on entry to `Safe`, then into `SunPointing` once `DETUMBLE_DURATION_TICKS` ticks have elapsed in `Safe`, checking the persisted `PANELS_DEPLOYED` parameter at that point to determine whether panel deployment is still needed	Inspection

Design

Component Type

Active component. Internal `Fw::Sm` state machine with four top level sibling states (`SAFE`, `DOWNLINK`, `SCIENCE`, `CHARGE`). `schedIn` is driven by `RateGroup2` (1 Hz).

Parameters

Parameter	Type	Description
`POWER_THRESHOLD`	`F32`	Minimum EPS state of charge (%) required for Downlink or Science
`EXPERIMENT_ENABLED`	`bool`	Ground set boolean enabling Science
`DOWNLINK_QUEUE_THRESHOLD`	`U32`	Minimum downlink queue depth (bytes) required to enter Downlink
`DETUMBLE_DURATION_TICKS`	`U32`	Ticks after entering `Safe` before commanding `AdcsApplication` from `Detumble` to `SunPointing`.
`PANELS_DEPLOYED`	`bool`	Set once solar panel deployment has completed. Read by `SatStateMachine` at the `DETUMBLE_DURATION_TICKS` boundary to determine whether deployment is still needed, so a completed deployment is not requested again.

Ports

Port	Direction	Type	Purpose
`schedIn`	Input	`Svc.Sched`	RateGroup2 (1 Hz) tick. Drives mode evaluation.
`orbitStateIn`	Input (async)	`Sat.OrbitStateInPort`	Flag for being over the ground station, from `GnssManager`
`downlinkQueueDepthIn`	Input (async)	`Sat.DownlinkQueueDepthInPort`	Current downlink queue depth in bytes, from `ComQueue`
`commsReadyIn`	Input (async)	`Sat.CommsReadyInPort`	Downlink readiness/permission flag from `ComApplication`; gates `DOWNLINK` entry.
`powerStateGet`	Output (sync get)	`Sat.PowerStateGetPort`	Pulls the latest power state from `EPSApplication` each tick
`pingIn` / `pingOut`	Input / Output	`Svc.Ping`	Health monitoring
`adcsModeOut`	Output	`Sat.AdcsModePort`	Mode command to `AdcsApplication`
`dataColModeOut`	Output	`Sat.DataColModePort`	Mode command to `DataCollectionApplication`
`scienceInferenceModeOut`	Output	`Sat.ScienceInferenceModePort`	Mode command to `ScienceApplication`
`commsModeOut`	Output	`Sat.CommsModePort`	Mode command to `ComApplication`
`thermalModeOut`	Output	`Sat.ThermalModePort`	Mode command to `ThermalApplication`
`prmGet`	Output	`Fw.PrmGet`	Load parameters from PrmDb
`logOut`	Output	`Fw.Log`	Event logging
`tlmOut`	Output	`Fw.Tlm`	Telemetry

Commands

Command Name	Parameters	Description	Postcondition
SAFE_MODE	—	Forces immediate transition to SAFE from any state	SatStateMachine is in SAFE; AdcsApplication commanded into Detumble
SAFE_EXIT	—	Requests transition to Charge	If checkoutComplete is set transitions to CHARGE. Otherwise denied, remains in SAFE
CHECKOUT_COMPLETE	—	Ground commands subsystem verification and OKs the transition	checkoutComplete latch is set
RESET	—	Forces reentry to SAFE	SatStateMachine is in SAFE; detumbleTickCounter reset to 0

State Machine

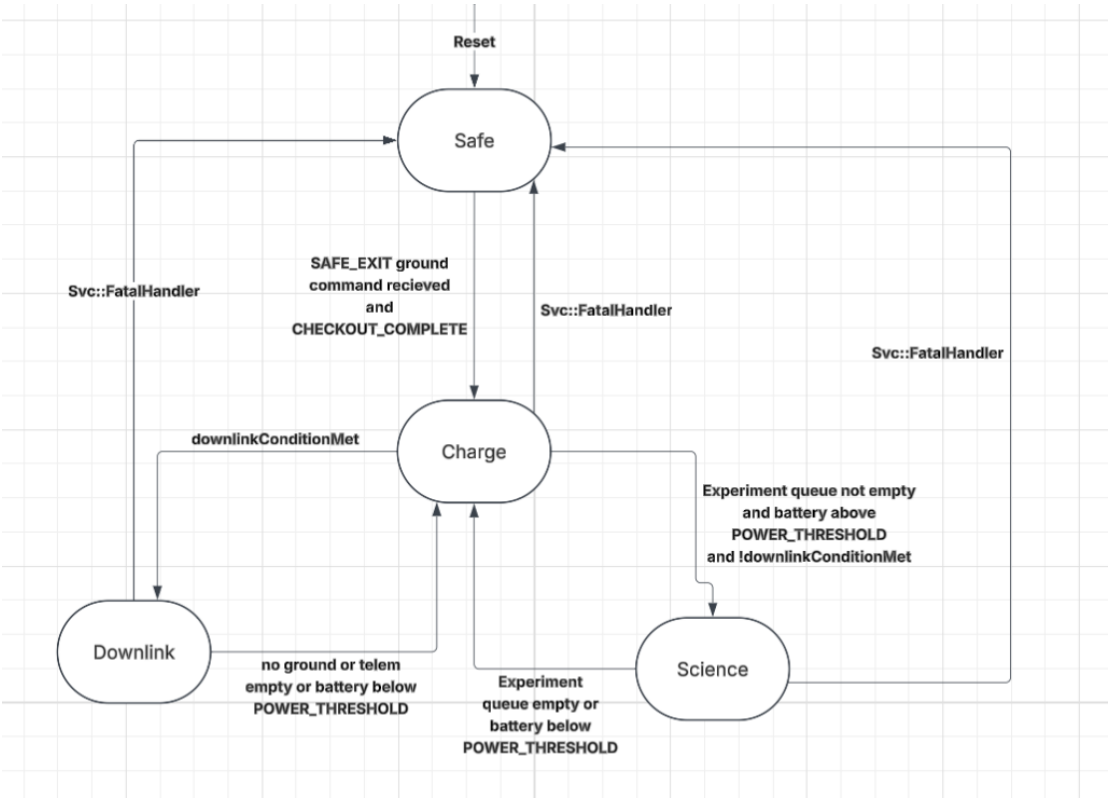


Figure 4.1: SatStateMachine mode transitions

Diagram terms:

Term	Meaning
SAFE_EXIT ground command received and CHECKOUT_COMPLETE	The SAFE_EXIT command is only honored once the CHECKOUT_COMPLETE ground command has latched the checkoutComplete flag.
downlinkConditionMet	True when all of: over the ground station (orbitStateIn), downlink queue depth above DOWNLINK_QUEUE_THRESHOLD (downlinkQueueDepthIn), state of charge above POWER_THRESHOLD, and ComApplication reports link readiness (commsReadyIn).
POWER_THRESHOLD	EPS state of charge parameter “Battery ok” / “battery above POWER_THRESHOLD” means the last powerStateGet reading is above this value.
Experiment queue not empty	Depth of the pending-experiment queue.

SAFE, DOWNLINK, SCIENCE, and CHARGE are flat top-level sibling states. SAFE is the plain safe state, reached from DOWNLINK/SCIENCE/CHARGE on the SAFE_MODE ground command, on startup, or after any reset. The 1 Hz priority evaluation is defined identically on DOWNLINK, SCIENCE, and CHARGE; SAFE does not run it.

```
SAFE
  entry: log SatModeSafeEntered
         reset detumbleTickCounter to 0
         command AdcsApplication Detumble
  exit:  log SatModeSafeExited
  on tick: cache condition inputs
           increment detumbleTickCounter
           if detumbleTickCounter == DETUMBLE_DURATION_TICKS:
             command AdcsApplication SunPointing
             if PANELS_DEPLOYED parameter is not set: flag panel deployment
               as still required
  on groundCommand(SAFE_MODE): reenter SAFE
  on groundCommand(SAFE_EXIT): enter CHARGE only if checkoutComplete is set,
                               else remain in SAFE
  on reset: reenter SAFE

DOWNLINK / SCIENCE / CHARGE
  entry: log SatMode<Name>Entered + command Layer 3 apps
  exit:  log SatMode<Name>Exited
  on groundCommand(SAFE_MODE): enter SAFE
  on groundCommand(SAFE_EXIT): ignored (already out of Safe;
                               does not disrupt the active mode)
  on reset: enter CHARGE
  on tick (defined identically on all three):
    cache condition inputs, then evaluate in priority order:
      1. if downlinkConditionMet (over ground station AND
         queue depth > DOWNLINK_QUEUE_THRESHOLD AND power OK
         AND commsReady)                                    -> DOWNLINK
      2. else if scienceConditionMet (power OK AND
         EXPERIMENT_ENABLED)                                -> SCIENCE
      3. else                                               -> CHARGE
    (low power alone never forces SAFE; it just fails the "power OK"
     clause above and falls through to CHARGE)
    (no-op, no exit/re-entry logged, if the winning state is the
     one already active)
```

Mode to app translation table

Mode	Safe	Downlink	Science	Charge
`AdcsApplication`	Detumble	AntennaPointing	EarthLimbPointing	SunPointing
`DataCollectionApplication`	Off	Off	RunExperiment	Off
`ScienceApplication`	Off	Off	ProcessImages	Off
`ComApplication`	Beacon	StandardDownlink	Beacon	Beacon
`ThermalApplication`	NoHeating	ActiveHeating	ActiveHeating	ActiveHeating

Hardware in each Mode

Mode	Safe	Charge	Downlink	Science
ADCS				
IMMU	1	1	1	1
SunSensor	1	1	1	1
Magnetorquer	1	1	1	1
StarTracker	0	0	1	1
Payload				
GNSS	0	0	0	1
LOST Camera	0	0	0	1
FOUND Camera	0	0	0	1
Comms				
Endurosat Radio	Beacon	Beacon	Downlink	Beacon
EPS				
Current Sensors	1	1	1	1
Panel Deploy	1*	0	0	0
Battery	1	1	1	1
Thermal				
Temperature Sensors	1	1	1	1
Heaters	0	1	1	1

'*' indicates that the panel deployment is not always on in safe mode but that that is the only time that could happen

Notes

- Reference: FPP flat/hierarchical state machines, choice pseudostates, inherited transitions

5 Svc::CmdDispatcher

Overview

Type: Active component.

`Svc::CmdDispatcher` decodes incoming command buffers and dispatches registered commands to the components that implement them. Ground and sequenced commands enter the topology through this shared command-and-data-handling path.

Command Dispatch, Validation, and Responses

At deployment startup, every command-capable component registers its opcodes with `Svc::CmdDispatcher`. The dispatcher maps each registered opcode to the output port connected to its implementing component. When an uplinked or sequenced command arrives, the dispatcher decodes the command buffer, checks that the opcode is registered, assigns a command sequence number, records the command context in its pending-command tracker, and sends the command to the registered component.

`Svc::CmdDispatcher` rejects a malformed command buffer, an unregistered opcode, or a command that cannot be tracked because the pending-command table is full. It logs the resulting condition and returns the applicable status to the command source through the corresponding response path. A full dispatcher input queue is handled by dropping the new buffer and recording an overflow condition rather than blocking the command path. In the HuskySat-2 flight topology, the ground-uplink path is configured as a command source, so the command result is returned through the ground command-response path for downlink and operator visibility.

The component that implements a command owns semantic validation, such as argument ranges, hardware readiness, and whether the requested action is legal in its current operational mode. It reports completion or rejection to `Svc::CmdDispatcher`, which correlates that response with the original command context and returns it to the source. This separates common transport and routing validation from component-specific safety decisions.

Relationship to SatStateMachine

Commands do not pass through `SatStateMachine` by default. From a systems perspective, this is justified because every command either leaves the satellite state unchanged or directly modifies the state contained in `SatStateMachine`. Additionally, routing every command through the state machine would duplicate F' opcode registration, response tracking, and component-level validation. `SatStateMachine` is the owner of satellite-wide operating-mode commands, so those commands are dispatched directly to its registered handlers. Other commands are dispatched directly to the application, manager, or service that owns the action; that handler checks its own state and interlocks before acting. This preserves F's command architecture while keeping satellite-wide mode authority in `SatStateMachine`.

F' Reference

CmdDispatcher software design document

6 F' System Services

These F' components provide deployment-wide telemetry, health, scheduling, and version services. Their individual descriptions are short, so they are grouped here; `Svc::CmdDispatcher` and `Svc::FatalHandler` are documented in dedicated chapters because they contain additional mission-specific design detail.

6.1 Svc::EventManager

Type: Active component.

`Svc::EventManager` collects component events, applies event filtering, and packages events for downlink. Applications and hardware managers publish diagnostic, warning, and fatal events to this shared path.

EventManager software design document

6.2 Svc::TlmChan and Svc::TlmPacketizer

Type: Active components.

`Svc::TlmChan` stores the most recent telemetry-channel values for collection. `Svc::TlmPacketizer` groups telemetry channels into packets according to configured definitions and rate logic. `ComApplication` changes packet rate logic as the spacecraft communication mode changes.

TlmChan software design document
TlmPacketizer source and interface

6.3 Svc::Health

Type: Queued component.

`Svc::Health` pings active components, detects missed responses, and provides the framework health-monitoring path. Critical applications and hardware managers register their ping ports with the shared service.

Health software design document

6.4 Svc::RateGroupDriver and Svc::ActiveRateGroup

Type: Passive and active scheduling components.

`Svc::RateGroupDriver` divides the primary system tick into scheduled rate-group signals. `Svc::ActiveRateGroup` invokes configured components at those rates so control loops, telemetry collection, and background work can use separate schedules.

RateGroupDriver software design document
ActiveRateGroup source and interface

HuskySat-2 designates the primary rate groups as follows:

Rate group	Frequency	Role
`RateGroup1`	10 Hz	Fast ADCS sensing, estimation, and control work.
`RateGroup2`	1 Hz	Satellite-state evaluation and nominal application, manager, and housekeeping work.
`RateGroup3`	0.1 Hz	Lower-rate ADCS and payload work that does not require the faster control-loop cadence.

6.4.1 HuskySat-2 rate-group offsets

The PAY and ADCS deployments will use configured phase offsets for selected work within an active rate-group period. For example, PAY will phase camera triggers, camera measurements, and associated attitude or position sampling; ADCS will phase sensor reads, filter updates, and actuator commands. These offsets keep the relative timing of measurements and commands repeatable, reduce contention from simultaneous thread wake-ups, and limit the effect of Linux scheduler jitter on control and image-capture timing.

The exact offsets and allowable jitter will be determined through hardware and deployment timing tests. Components will use the rate-group tick together with available hardware or measurement timestamps, so an offset will provide scheduling guidance rather than replace a sensor timestamp. If work misses its expected phase or produces stale data, the owning application will detect that condition and will not treat the sample as current.

6.5 Svc::Version

Type: Passive component.

`Svc::Version` reports flight-software version and build information through the CDH telemetry path for checkout, GDS testing, and flight operations.

Version software design document

6.6 Svc::AssertFatalAdapter

Type: Passive component.

`Svc::AssertFatalAdapter` converts framework assertions into FATAL events for the CDH fault-handling path. Reset and recovery policy remains a system-level deployment decision.

AssertFatalAdapter software design document

7 Svc::FatalHandler

Overview

`Svc::FatalHandler` is an F' system-service component that receives fatal conditions and applies the deployment's terminal fatal-response policy. It is placed in the Level 4 system-services and control layer because its response is satellite-wide rather than specific to one hardware subsystem.

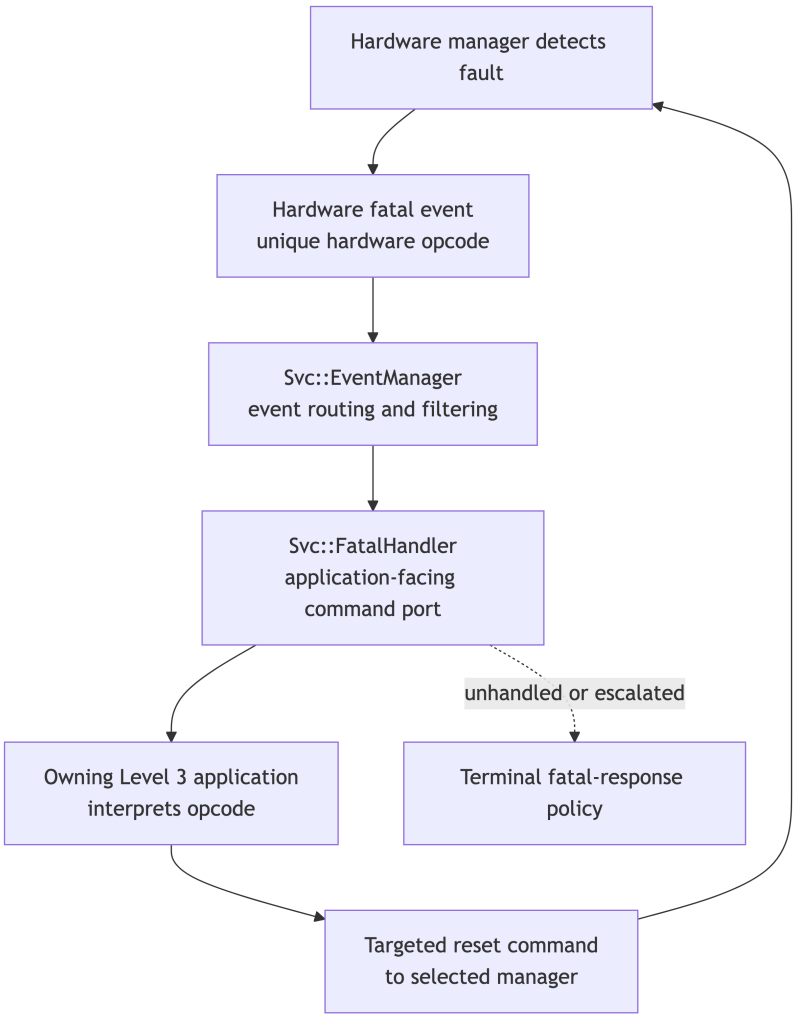
HuskySat-2 Use

Hardware managers remain responsible for detecting device-specific failures and reporting them through typed events. `Svc::FatalHandler` has an application-facing command port for each Level 3 application. When it receives a fatal notification, it uses the appropriate port to send an application command carrying the fatal condition to the owning application. A hardware-related fatal event carries a unique opcode identifying the affected hardware; the application maps that opcode to the appropriate manager and issues that manager's reset command. This preserves the App-Manager-Driver boundary: the application decides what recovery action is appropriate, the manager owns the device-specific reset sequence, and the driver only exposes the low-level interface.

`Svc::FatalHandler` will remain the final system-level path for unhandled or escalated fatal conditions. It does not replace a hardware manager or become a central hardware-reset owner. If an event cannot be mapped safely to an owning application or hardware opcode, the deployment will escalate it to the terminal fatal-response policy.

Proposed Hardware-Fault Pipeline

The following is a proposed event and recovery flow. The exact event ports, opcode type, and escalation thresholds are TBR.



This design keeps reset authority with the application-manager path while giving the system a defined terminal response when targeted recovery is unsafe or unavailable. The opcode namespace will be centrally documented so that every hardware manager emits unique, stable identifiers.

F' Reference

FatalHandler software design document

Part II

Level 3 - Application Layer

8 F' Application and Data Services

These F' components support the Level 3 application and data paths. They are grouped here so that short framework-component descriptions do not each force a new PDF page.

8.1 Svc::ComQueue

Type: Active component.

`Svc::ComQueue` queues event, telemetry, and other outgoing packets for transmission through the communications topology. It decouples packet producers from radio timing and framing while `ComApplication` controls the mission communication mode around the shared path.

ComQueue source and interface

8.2 Svc::FrameAccumulator

Type: Passive component.

`Svc::FrameAccumulator` accumulates incoming bytes until a complete communications frame is available for deframing. It handles stream boundaries without owning command interpretation.

FrameAccumulator source and interface

8.3 Svc::FprimeRouter

Type: Passive component.

`Svc::FprimeRouter` routes decoded packets to the appropriate command, telemetry, or event destination. It connects the communications receive path to `Svc::CmdDispatcher` and other packet consumers after uplink frames have been deframed.

FprimeRouter source and interface

8.4 Svc::FprimeFramer and Svc::FprimeDeframer

Type: Passive components.

`Svc::FprimeFramer` wraps outgoing packets in native F' framing. `Svc::FprimeDeframer` removes native F' framing from received packets before command routing.

FprimeFramer source and interface
FprimeDeframer source and interface

8.5 Svc::Ccsds::TmFramer and Svc::Ccsds::TcDeframer

Type: Passive components.

`Svc::Ccsds::TmFramer` formats outgoing telemetry and event packets into CCSDS telemetry frames. `Svc::Ccsds::TcDeframer` validates and removes CCSDS telecommand framing from received frames before space-packet processing.

CCSDS TM framer source
CCSDS TC deframer source

8.6 Svc::Ccsds::SpacePacketFramer and Svc::Ccsds::SpacePacketDeframer

Type: Passive components.

`Svc::Ccsds::SpacePacketFramer` constructs CCSDS space packets from outgoing F' packet data. `Svc::Ccsds::SpacePacketDeframer` validates and extracts CCSDS space packets from received telecommand frames.

CCSDS space-packet framer source
CCSDS space-packet deframer source

8.7 Svc::Ccsds::ApidManager

Type: Passive component.

`Svc::Ccsds::ApidManager` manages CCSDS application-process identifiers and packet sequence information used by the communications path.

CCSDS APID manager source

8.8 Svc::ComAggregator

Type: Passive component.

`Svc::ComAggregator` combines outgoing packet streams before they enter the configured communications framing path while preserving packet boundaries.

ComAggregator source and interface

8.9 Svc::FileUplink and Svc::FileDownlink

Type: Active components.

`Svc::FileUplink` receives file-transfer packets and reconstructs files in the onboard filesystem. `Svc::FileDownlink` reads files from onboard storage and queues them for downlink. Together they support configuration, parameter, science-product, log, and diagnostic-file transfers.

FileUplink source and interface
FileDownlink source and interface

8.10 Svc::FileManager

Type: Active component.

`Svc::FileManager` provides commands and services for creating, deleting, inspecting, and otherwise managing files in the onboard filesystem.

FileManager source and interface

8.11 Svc::PrmDb

Type: Active component.

`Svc::PrmDb` stores and retrieves persistent component parameters. It provides the parameter-persistence path used to retain configured values across deployments and reboot cycles.

PrmDb source and interface

8.12 Svc::DpManager, Svc::DpWriter, and Svc::DpCatalog

Type: Active components.

`Svc::DpManager` allocates and manages data-product containers. `Svc::DpWriter` writes completed science and diagnostic products to persistent storage. `Svc::DpCatalog` catalogs products and manages their availability for later operations such as downlink.

DpManager source and interface
DpWriter source and interface
DpCatalog source and interface

9 AdcsApplication

Overview

`AdcsApplication` is the Layer 3 Active component for the ADCS subsystem. It owns the attitude control loop and switches operating mode on command from `SatStateMachine`. It dispatches actuator commands to Layer 2 hardware managers (`IMMUManager`, `SunSensorManager`, `MagnetorquerManager`) and consumes attitude from `StarTrackerManager` and timing from `GnssManager`.

Attitude estimation and all control law computation are internal to `AdcsApplication` itself. `AdcsApplication` receives raw sensor data directly from the hardware managers and `StarTrackerManager`, maintains its own cached state estimate, and runs internal control functions: `DetumbleController` in `Detumble`, `SlewLimiter` in the three maneuvering pointing modes, and `PointingController` in steady-state `AttitudeHold`. It then turns that estimate into a commanded magnetic moment.

ADCS software topology:

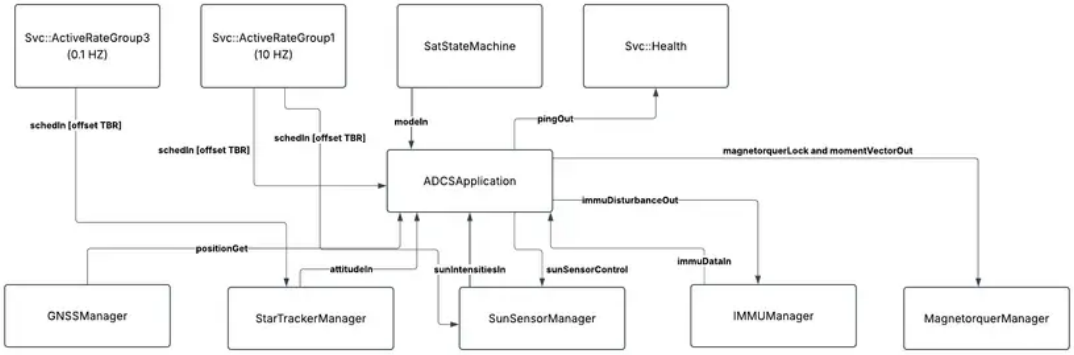


Figure 9.1: ADCS application topology diagram

Requirements

ID	Requirement	Verification
HS2-ADC-001	AdcsApplication shall detumble the spacecraft	Inspection
HS2-ADC-002	AdcsApplication shall point the spacecraft at a commanded target quaternion	Inspection
HS2-ADC-003	AdcsApplication shall maintain a slew rate under 0.04 degrees per second upon command	Inspection
HS2-ADC-004	AdcsApplication shall respond to health pings within the required deadline	Inspection
HS2-ADC-005	AdcsApplication shall assert WARNING_HI events followed by a FATAL if hardware is unable to recover from errors	Inspection
HS2-ADC-006	AdcsApplication shall answer IMMUManager's `configureDataRequest` with connection port name, accel, gyro, and magnetometer data rates, and startup wait time	Inspection
HS2-ADC-007	AdcsApplication shall lock MagnetorquerManager and assert the IMMUManager magnetic disturbance flag together, and unlock and clear them together	Inspection

ID	Requirement	Verification
HS2-ADC-008	AdcsApplication shall update its internal estimated angular rate on each received IMU sample	Inspection
HS2-ADC-009	AdcsApplication shall update its internal estimated attitude quaternion on each received star tracker attitude sample	Inspection
HS2-ADC-010	AdcsApplication shall update its internal sun vector estimate on each received sun vector sample	Inspection
HS2-ADC-011	AdcsApplication shall record the timestamp of each received measurement, per source (IMU, sun vector, star tracker)	Inspection
HS2-ADC-012	AdcsApplication shall retain the previous B-field measurement for use by `DetumbleController`'s derivative term	Inspection
HS2-ADC-013	AdcsApplication shall compute a magnetic moment vector from the current B-field and previous Bfield via `DetumbleController` (using B-dot law) while in `Detumble`	Inspection
HS2-ADC-014	AdcsApplication shall compute a magnetic moment vector from the estimated attitude, angular rate, target quaternion, and B-field via `PointingController` (quaternion PD control law) while in `AttitudeHold`	Inspection
HS2-ADC-015	AdcsApplication shall compute a magnetic moment vector via `SlewLimiter`, capping the commanded rotation to the active slew limit, while in `SunPointing`, `AntennaPointing`, or `EarthLimbPointing`	Inspection
HS2-ADC-016	AdcsApplication shall compute a body frame sun vector via `SunVectorEstimator` from the six calibrated intensities published by `SunSensorManager`	Inspection

Design

Component Type

Active component with internal hierarchical F' state machine (`Fw::Sm`).

Mode Interface

`AdcsApplication` receives its operating mode from `SatStateMachine` via a typed port:

```
sync input port modeIn: Sat.AdcsModePort   # carries Adcs.Mode
```

Mode enum (owned by this component's module):

```
module Adcs {
    enum Mode { Off, Detumble, SunPointing, AntennaPointing, EarthLimbPointing, AttitudeHold }
}
```

If the incoming mode matches the current mode, the handler returns immediately.

Ports

Port	Direction	Type	Purpose
`modeIn`	Input	`Sat.AdcsModePort`	Mode command from SatStateMachine
`schedIn`	Input	`Svc.Sched`	Rate group tick (10 Hz)

Port	Direction	Type	Purpose
`immuConfigureGet`	Input (sync)	`immuConfigurationParameters_p` → `Adcs.ConfigureParameters`	Answers IMMUManager's `configureDataRequest`
`immuDataIn`	Input (async)	`Adcs.ImmuDataPort`	IMMU angular rate + acceleration sample from `IMMUManager`; updates the internal angular rate estimate
`sunIntensitiesIn`	Input (async)	`Adcs.SunIntensityPort`	Six calibrated channel intensities + timestamp from `SunSensorManager` run through `SunVectorEstimator` to update the internal sun vector estimate
`attitudeIn`	Input (async)	`Adcs.AttitudePort`	Absolute attitude quaternion from `StarTrackerManager` that updates the internal attitude estimate
`sunSensorControl`	Output	`Adcs.ManagerControlPort`	ON/OFF command to SunSensorManager
`magnetorquerLock`	Output	`mtToggle_p` (`is_locked: bool`)	Lock/unlock command to MagnetorquerManager
`immuDisturbanceOut`	Output	`mtToggle_p` (`is_locked: bool`)	Magnetic disturbance flag to IMMUManager which is asserted/cleared together with `magnetorquerLock`
`momentVectorOut`	Output	`Adcs.MomentVectorPort`	Send computed moment vector to MagnetorquerManager
`positionGet`	Output	`Fw.Dp`	Request position/timing from GnssManager
`pingIn` / `pingOut`	In/Out	`Svc.Ping`	Health monitoring
`logOut`	Output	`Fw.Log`	Event logging
`tlmOut`	Output	`Fw.Tlm`	Telemetry

Internal State Estimate

Referred to as `AttitudeFilter` in the diagrams below (its name when it was a separate component) — internal to `AdcsApplication` now, not a distinct component.

Field	Updated by	Description
Estimated attitude quaternion	`attitudeIn`	Most recent absolute attitude from `StarTrackerManager`
Estimated angular rate	`immuDataIn`	Most recent angular rate from `IMMUManager`
Estimated angular acceleration	`immuDataIn`	Difference of consecutive angular rate samples over the tick period. `SlewLimiter` needs this in addition to rate
Sun vector estimate	`sunIntensitiesIn`, via `SunVectorEstimator`	Sun vector computed from the six calibrated intensities
Current & previous B-field	`immuDataIn`	Current and prior B-feild, retained for `DetumbleController`'s derivative term
Timestamps	each of the above	Last received time for IMU, sun vector, and star tracker samples, used to judge data freshness

9.0.0.1 Kalman filter

- **State vector:** attitude quaternion `q`, angular velocity (3), gyro bias `b_g` (3), 9-element error-state covariance (`Matrix<9>`) over attitude-error/rate/bias. Parameters are `sigma_gyr` (gyro noise), `sigma_omega_rw` (rate random walk), `sigma_bg_rw` (bias random walk), `sigma_st` (star tracker noise)

9-state MEKF Attitude error (3) + angular velocity (3) + gyro bias (3) as filter states. Gyro readings treated as a noisy *measurement* of a separately propagating rate state (Kalman corrected each tick), not fed straight into propagation.

Prediction created by propagating quaternion via kinematics driven by the current rate estimate, propagating the 9×9 covariance with process noise `sigma_omega_rw`/`sigma_bg_rw`. Star tracker gives a full quaternion correction (`sigma_st`), gyro gives a rate correction (`sigma_gyr`).

Internal Control Functions

Three control law computations are internal functions inside `AdcsApplication`.

Function	Inputs	Output	Called from
`DetumbleController`	Current B-field, previous B-field	Magnetic moment vector computed from the change in the B-field using the B-dot law.	`DETUMBLE/RUNNING`, each tick. The only control function active in `Detumble`
`SlewLimiter`	Estimated attitude, angular rate, angular acceleration, target quaternion, B-field, slew rate limit	Magnetic moment vector, via a cascaded control law. Outer P loop on the attitude error angle (capped to the slew limit) produces a rate command, inner rate PID loop (using angular acceleration as its derivative term) turns that into a torque, then computes the magnetic moment as the cross product of B and torque divided by the squared magnitude of B.	`SUN_POINTING`, `ANTENNA_POINTING`, and `EARTH_LIMB_POINTING`'s `TRACKING` substate
`PointingController`	Estimated attitude, angular rate, target quaternion, B-field	Magnetic moment vector. Quaternion PD law computes torque from attitude/rate error against the target, converted to moment the same way as `SlewLimiter`	`ATTITUDE_HOLD`'s `HOLDING` substate

Commands

Mnemonic	Args	Description
`RESET`	—	Force transition back to current mode's initial substate

State Machine

`AdcsApplication` uses a hierarchical F' state machine. A single `switchMode: Adcs.Mode` signal defined at the top level is inherited by all leaf states, making mode switches valid from any nested substate.

```
OFF
  entry: send ManagerControl.OFF via sunSensorControl
         call magnetorquerLock(true) and immuDisturbanceOut(true)
  exit:  send ManagerControl.ON  via sunSensorControl
         call magnetorquerLock(false) and immuDisturbanceOut(false)

DETUMBLE
    RUNNING
       on tick: read cached current/previous B-field
                DetumbleController(currentBField, previousBField) → momentVector
                send momentVectorOut

SUN_POINTING
    ACQUIRING       (on tick: check cached sun-vector freshness; wait for valid sun vector)
```

```
    TRACKING
       on tick: read cached attitude, angular rate, angular acceleration, B-field, sun-vector
        ↪ freshness
                SlewLimiter(attitude, angularRate, angularAccel, targetQuaternion, bField, 0.04°/s)
                 ↪ → momentVector
                send momentVectorOut

ANTENNA_POINTING
    ACQUIRING       (on tick: compute ground station quaternion via GNSS ephemeris)
    TRACKING
       on tick: read cached attitude, angular rate, angular acceleration, B-field
                SlewLimiter(attitude, angularRate, angularAccel, targetQuaternion, bField, 0.04°/s)
                 ↪ → momentVector
                send momentVectorOut

EARTH_LIMB_POINTING
    ACQUIRING       (on tick: acquire lit Earth limb target + solar optimization)
    TRACKING
       on tick: read cached attitude, angular rate, angular acceleration, B-field, star-tracker
        ↪ freshness
                SlewLimiter(attitude, angularRate, angularAccel, targetQuaternion, bField, 0.04°/s)
                 ↪ → momentVector
                send momentVectorOut

ATTITUDE_HOLD
    HOLDING
       on tick: read cached attitude, angular rate, B-field
                PointingController(attitude, angularRate, targetQuaternion, bField) → momentVector
                send momentVectorOut

# Inherited by all leaf states:
on switchMode(Adcs.Mode.Off)               enter OFF
on switchMode(Adcs.Mode.Detumble)          enter DETUMBLE
on switchMode(Adcs.Mode.SunPointing)       enter SUN_POINTING
on switchMode(Adcs.Mode.AntennaPointing)   enter ANTENNA_POINTING
on switchMode(Adcs.Mode.EarthLimbPointing) enter EARTH_LIMB_POINTING
on switchMode(Adcs.Mode.AttitudeHold)      enter ATTITUDE_HOLD
```

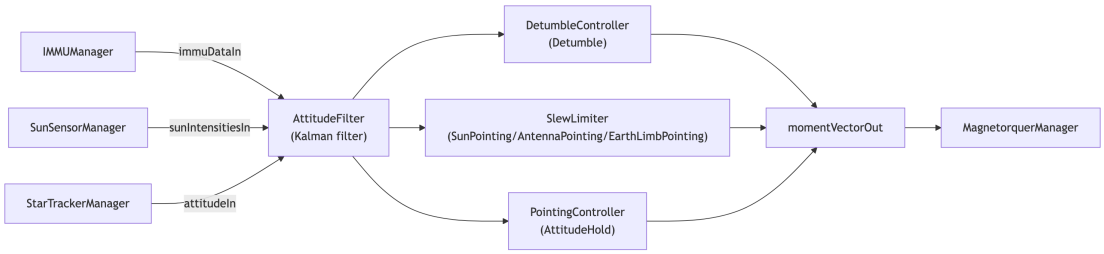
`switchMode` is inherited by every leaf state, so any of the 6 modes can reach any other directly, not just from `OFF`. On entry to `OFF`, `AdcsApplication` locks `MagnetorquerManager` and asserts `ImmuManager`'s disturbance flag; `DETUMBLE` alone runs `bDotCompute`, while `SUN_POINTING`, `ANTENNA_POINTING`, `EARTH_LIMB_POINTING`, and `ATTITUDE_HOLD` all run the same `slewRateCompute` + `quaternionPdCompute` → `momentVectorOut` pipeline each tick, differing only in target source.

`SUN_POINTING`, `ANTENNA_POINTING`, and `EARTH_LIMB_POINTING` also each nest the same `ACQUIRING` → `TRACKING` substate pattern, moving to `TRACKING` once that mode's target is acquired: `SUN_POINTING` waits for a valid sun vector; `ANTENNA_POINTING` computes the ground station quaternion from GNSS ephemeris; `EARTH_LIMB_POINTING` acquires the lit Earth limb target. `DETUMBLE` and `ATTITUDE_HOLD` each have a single unconditional substate (`RUNNING`, `HOLDING` respectively) with no internal transition.

Reference: FPP inherited transitions, FPP substates

Notes

Internal control flow:


IMMUManager
immuDataIn
SunSensorManager
sunIntensitiesIn
StarTrackerManager
attitudeIn
AttitudeFilter
(Kalman filter)
DetumbleController
(Detumble)
SlewLimiter
(SunPointing/AntennaPointing/EarthLimbPointing)
PointingController
(AttitudeHold)
momentVectorOut
MagnetorquerManager

10 DataCollectionApplication

`DataCollectionApplication` is the Layer 3 active component for the DataCollection subtopology. Every image capture is a ground-commanded event: ground sends `QUEUE_EXPERIMENT` with a complete set of experiment parameters.

Data Collection software topology:

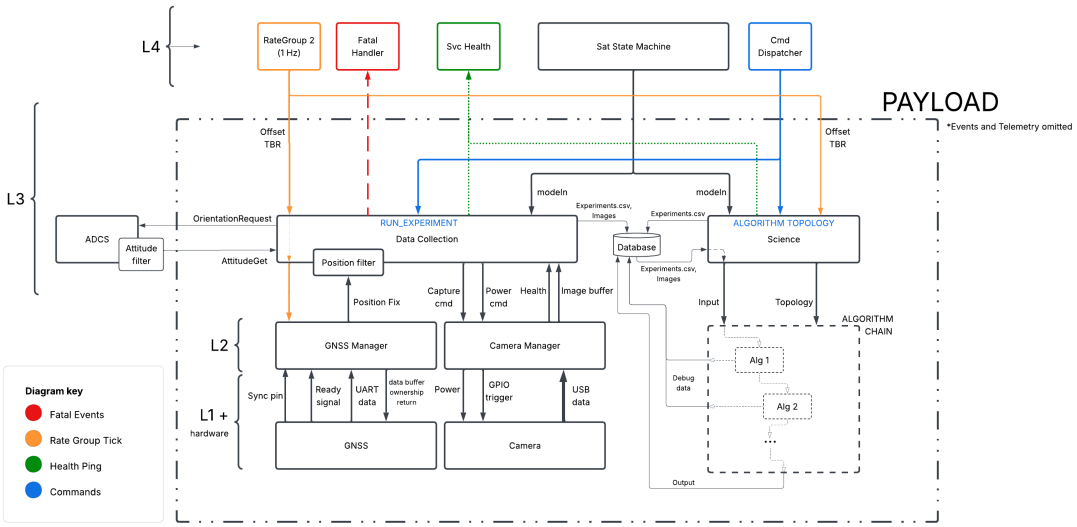


Figure 10.1: Data Collection application topology diagram

File system
The partition holds two things: a manifest, and one subdirectory per `Science.ImageType` integer value. `<imagePartitionDir>/experiments.csv` logs imaging opportunities.

Each experiment opportunity has:

- `timeRequested` as `YYYY:MM:DD:HH:MM:SS`,
- `timeDataCollected` as `YYYY:MM:DD:HH:MM:SS`,
- `timeScienceStarted` as `YYYY:MM:DD:HH:MM:SS`,
- `timeScienceFinished` as `YYYY:MM:DD:HH:MM:SS`,
- `positionKnown` as `bool`,
- `position` as `x:y:z`
- `attitude` as `x:y:z:w` (quaternion),
- `availableImageTypes` a `U16` bitmask representing `Science.ImageType`
- `experimentID` as a `U16`

`<imagePartitionDir>/1/` holds `STARS` images, `<imagePartitionDir>/2/` holds `HORIZON` images, etc.
`fileName` is the experiment ID

The component is driven by two synchronous input ports:

- `schedIn` (`Svc.Sched`) — a rate-group tick, used only for `pingIn`/`pingOut` liveness and polling camera status
- `modeIn` (`DataCollection.DataColModePort`, carrying `Types.DataColMode`) — mode commands from `SatStateMachine`

Requirements

ID	Requirement	Verification
HS2-DCA-001	DataCollectionApplication shall capture data only in response to a ground-issued `QUEUE_EXPERIMENT` command, using that command's own arguments — never automatically or on a schedule	Inspection
HS2-DCA-002	DataCollectionApplication shall power on both cameras prior to a commanded experiment	Inspection
HS2-DCA-003	DataCollectionApplication shall perform a health check on both cameras before running an experiment and reject the command with `WARNING_HI` if any check fails	Inspection
HS2-DCA-004	DataCollectionApplication shall simultaneously acquire images, attitude, and position within 50ms of each other	Inspection
HS2-DCA-005	DataCollectionApplication shall write each captured image into Science's image partition under its `Types.ImageTyp e`/camera-prefix path, and append one row describing the opportunity to `experiments.csv`	Inspection
HS2-DCA-006	DataCollectionApplication shall reject a `QUEUE_EXPERIMENT` command with `WARNING_HI` and take no images if the image-partition mount is at or above its configured full threshold	Inspection
HS2-DCA-007	DataCollectionApplication shall power off both cameras upon experiment completion or error	Inspection
HS2-DCA-008	DataCollectionApplication shall report experiment success or failure upon completion	Inspection
HS2-DCA-009	DataCollectionApplication shall switch operating mode on command from SatStateMachine	Inspection
HS2-DCA-010	DataCollectionApplication shall respond to health ping within the required deadline	Inspection

Ports

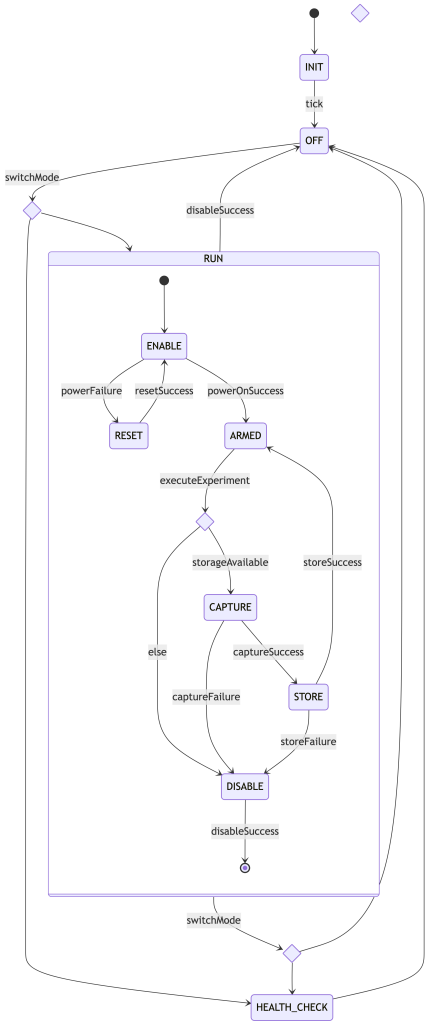
Port	Direction	Type	Purpose
`modeIn`	Input	`DataCollection.DataCol ModePort`	Mode command from SatStateMachine, carrying `Types.Data ColMode`.
`schedIn`	Input	`Svc.Sched`	Rate group tick; drives `pingIn`/`pingOu t` and the state machine's `tick` signal
`cameraPower[2]`	Output	DataCollection.Camer-aPower(camera, powerOn); returns Types.CameraStatus	Power `lostCameraManager`/`foundCame raManager` (port index 0/1) on or off.
`cameraCheckup[2]`	Output	DataCollection.Camer-aCheckup(camera); returns Types.CameraStatus	Per-camera health check, used by `Heal thCheck` mode's `checkup` action.
`cameraCapture[2]`	Output	DataCollection.Camera-Capture(camera, imageType); returns Types.CameraStatus	Triggers one camera's capture of the current experiment's `imageType`.
`positionGet`	Output	DataCollection.Posi-tionGet(); returns Types.PositionData	Requests the current estimated position vector
`attitudeGet`	Output	DataCollection.Atti-tudeGet(); returns Types.Quaternion	Requests the current estimated attitude quaternion.
`OrientationRequest`	Output	DataCollection.Orienta-tionRequest(); returns Types.Quaternion	Requests a satellite orientation
`pingIn` / `pingOut`	In/Out	`Svc.Ping`	Health monitoring.

Commands

Name	Args	Description
`QUEUE_EXPERIMENT`	expId (U8); imageTypeCode (Types.ImageTypes); count (U8)	Adds a new entry into experiments.csv to queue an experiment. expId parameter reserves all ids from expId to expId+(count-1)

Once an experiment is commanded, Data Collection calculates and requests an attitude from ADCS based on the required image types. Data Collection then polls the current attitude until the request is satisfied. If no attitude can satisfy the requirements at the current position in orbit, a warning is thrown and Data Collection caches the request to keep working through the queue.

State Machine




```
[*] --> INIT
INIT --> OFF : tick

OFF
  entry: declare
  on switchMode --> OFF_SWITCH_MODE (choice)
    if modeIsHealthCheck --> HEALTH_CHECK
    else                 --> RUN
```

```
HEALTH_CHECK
  entry: powerOn
  on powerFailure:   powerOff, --> OFF
  on checkupSuccess: powerOff, --> OFF
  on checkupFailure: powerOff, --> OFF
  on switchMode:     powerOff, --> HEALTH_CHECK_SWITCH_MODE (choice)
    if modeIsOff --> OFF
    else         --> RUN

RUN
  initial --> ENABLE
  on switchMode: powerOff, --> RUN_SWITCH_MODE (choice)   [inherited by every substate below]
    if modeIsOff --> OFF
    else         --> HEALTH_CHECK

  RESET
    entry: powerOff
    on resetSuccess --> ENABLE

  ENABLE
    entry: powerOn
    on powerOnSuccess --> ARMED
    on powerFailure   --> RESET

  ARMED
    entry: (none beyond declare)
    on executeExperiment --> STORAGE_CHECK

  STORAGE_CHECK (choice)
    if storageAvailable --> CAPTURE
    else                --> DISABLE

  CAPTURE
    entry: captureData
    on captureSuccess --> STORE
    on captureSlow    --> CAPTURE
    on captureFailure --> DISABLE

  STORE
    entry: storeFiles
    on retryStore   --> STORE
    on storeSuccess --> ARMED
    on storeFailure --> DISABLE

  DISABLE
    entry: declare, powerOff
    on disableSuccess --> OFF
```

11 ComApplication

Overview

`ComApplication` is the Layer 3 Active component for the Comms subtopology. It manages the EnduroSat S-band radio operating mode, switching between `BEACON` to transmit real-time SOH telemetry at 1Hz, `STANDARD_DOWNLINK` mode to transmit all real-time telemetry including payload experiment data, `STORED_PLAYBACK` mode to downlink stored and real-time SOH telemetry concurrently, and `NO_DOWNLINK` to cease transmission upon command.

Communications software topology:

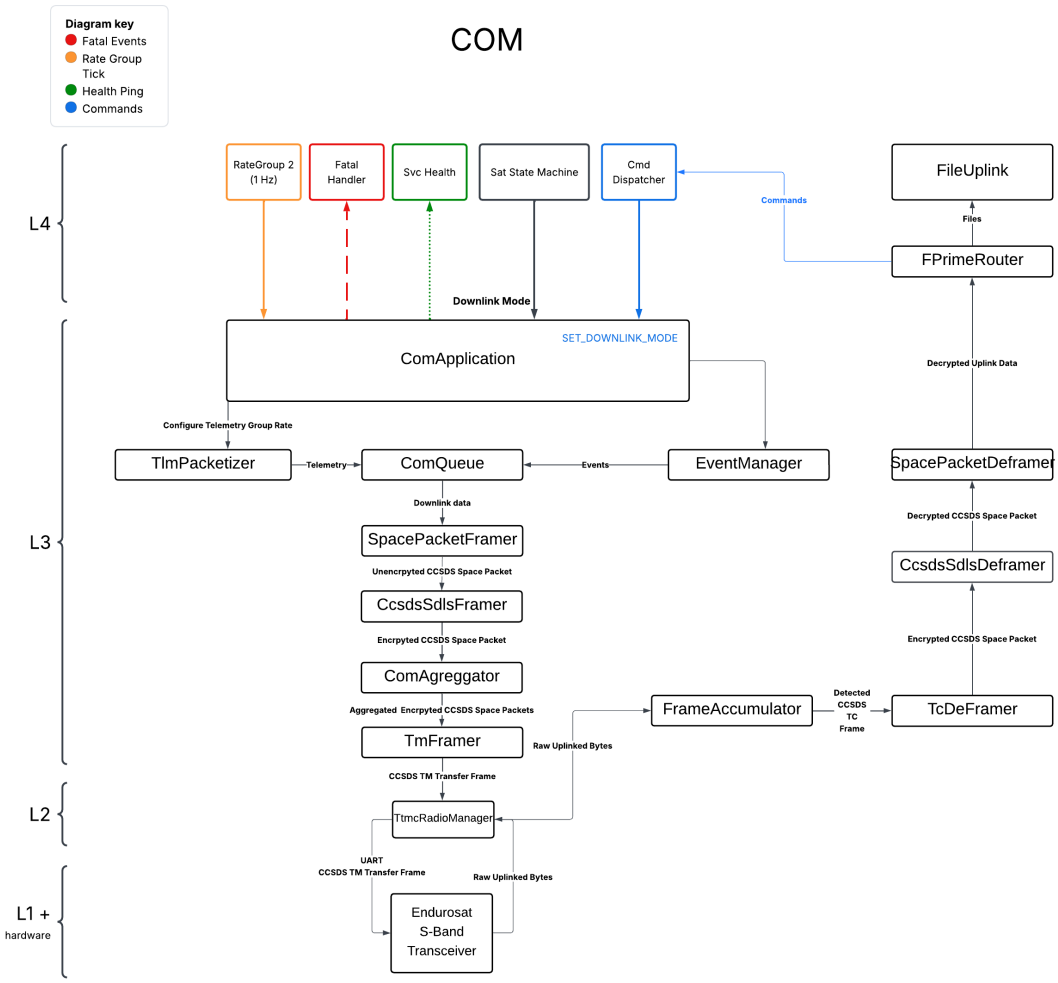


Figure 11.1: Communications application topology diagram

Terminology

Term	Definition
Telemetry	Information that is downlinked to the HS-2 ground station including: F' events, images/files, and data packets

Requirements

ID	Requirement	Verification
HS2-COM-001	ComApplication shall activate `BEACON` downlink mode when commanded by `SatStateMachine`	Inspection
HS2-COM-002	ComApplication shall activate `STANDARD_DOWNLINK` downlink mode when commanded by `SatStateMachine`	Inspection
HS2-COM-003	ComApplication shall activate `STORED_PLAYBACK` downlink mode only when commanded	Inspection
HS2-COM-004	ComApplication shall respond to health ping within the required deadline	Inspection
HS2-COM-005	ComApplication shall enable the transmission of stored telemetry along with real-time telemetry when `STORED_PLAYBACK` downlink mode is activated	Inspection
HS2-COM-006	ComApplication shall enable the transmission of SOH telemetry only at a 1Hz rate when `BEACON` downlink mode is activated	Inspection
HS2-COM-007	ComApplication shall enable the transmission of all real-time telemetry when `STANDARD_DOWNLINK` downlink mode is activated	Inspection
HS2-COM-008	ComApplication shall disable all transmission when `NO_DOWNLINK` mode is actived	Inspection

Design

Component Type

Active component with internal hierarchical F' state machine (`Fw::Sm`).

Mode Interface

`ComApplication` receives its operating mode from `SatStateMachine` via a typed port:

```
sync input port modeIn: Sat.CommsModePort   # carries Comms.Mode
```

Mode enum (owned by this component's module):

```
module Com {
    enum Mode { BEACON, STANDARD_DOWNLINK, STORED_PLAYBACK, NO_DOWNLINK }
}
```

If the incoming mode matches the current mode, the handler returns immediately (idempotent).

BEACON Mode

During this mode, only SOH telemetry shall be transmitted. This mode is intended for the satellite to establish a connection with the ground station, particularly when detumbling or when outside of a communication window. In BEACON mode, the satellite shall only transmit a single SOH telemetry packet at 1Hz to ensure minimum power draw.

This mode will set all packets, except SOH, in the TlmPacketizer to have a RateLogic of SILENCED.

This mode will be commanded by the SatStateMachine, but the ground operators can set this through the SET_DOWNLINK_MODE command.

STANDARD_DOWNLINK mode

During this mode, all real-time telemetry shall be downlinked to the ground station at 1Hz.

This mode will set all packets, except SOH, in the TlmPacketizer to have a RateLogic of ON_CHANGE_MIN.

This mode will be commanded by the SatStateMachine, but the ground operators can set this through the SET_DOWNLINK_MODE command.

STORED_PLAYBACK mode

During this mode, all stored telemetry, which includes payload experiment data, will be downlinked to the ground station. The priority in this mode is to downlink images and payload data captured during experiments. Real-time SOH telemetry will be downlinked at a rate of 1 Hz.

This mode will set SOH telemetry in the TlmPacketizer to have a RateLogic of EVERY_MAX to ensure stored telemetry is given priority. The Svc::DpCatalog component can be used to downlink generated data products (such as images and stored telemetry) that exist within a specified set of directories.

This mode will be commanded by the ground operators through the SET_DOWNLINK_MODE command.

NO_DOWNLINK mode

During this mode, nothing will be transmitted from the satellite. Refer to requirement UNP12-91 within the RVM.

This mode will be commanded by the ground operators through the SET_DOWNLINK_MODE command.

Ports

Port	Direction	Type	Purpose
modeIn	Input	Sat.CommsModePort	Mode command from SatStateMachine
schedIn	Input	Svc.Sched	Rate group tick
pingIn	Input	Svc.Ping	Health monitoring input ping from Svc::Health to component
pingOut	Output	Svc.Ping	Health monitoring ping response from component to Svc::Health
logOut	Output	Fw.Log	Event logging
tlmOut	Output	Fw.Tlm	Telemetry (current mode) that will be used in SOH packets
configureGroupRate	Output	Svc.ConfigureGroupRate	Configure rate at which certain telemetry sections are transmitted as per the Svc.TlmPacketizer component

ComApplication Telemetry

The following telemetry will be used for the `ComApplication`:

Mnemonic	Type	Description
`downlink_mode`	`Comms.Mode`	Downlink mode that will be set by `SatStateMachine`. Can be one of the following values: `BEACON`, `STORED_PLAYBACK`, `STANDARD_DOWNLINK`, or `NO_DOWNLINK`

ComApplication Events

The following events will be emitted by the `ComApplication`:

Event Name	Severity Level	Description
DownlinkModeUpdateCmd	COMMAND	Emitted when ComApplication processes a command from the ground to update downlink mode
DownlinkModeSatStateUpdate	ACTIVITY_HI	Emitted when `SatStateMachine` updates the link mode through the `modeIn` input port

Commands

Command Name	Parameters	Description	Postcondition
SET_DOWNLINK_MODE	downlink_mode: `Com.Mode`	Sets the current downlink mode to `downlink_mode`	`ComApplication.downlink_mode` will be changed to `downlink_mode` and `TlmPacketizer` will have telemetry section rates updated as per `ComApplication 3.2`

State Machine

`ComApplication` uses a hierarchical F' state machine with the following states: `BEACON`, `STORED_PLAYBACK`, `STANDARD_DOWNLINK`, `NO_DOWNLINK`. All states will only be commanded by the `SatStateMachine` as the `ComApplication` does not follow a traditional state-machine like a hardware-manager.

```
BEACON
  entry: Set all packets, except SOH, in the `TlmPacketizer` to have a `RateLogic` of `SILENCED`.
   ↪ Set `downlinkMode` telemetry to be `BEACON`.

STORED_PLAYBACK
  entry: Set SOH telemetry in the `TlmPacketizer` to have a `RateLogic` of `EVERY_MAX`. Set
   ↪ `downlinkMode` telemetry to be `STORED_PLAYBACK`.

STANDARD_DOWNLINK
  entry: set all packets, except SOH, in the `TlmPacketizer` to have a `RateLogic` of
   ↪ `ON_CHANGE_MIN`. Set `downlinkMode` telemetry to be `STANDARD_DOWNLINK`.

NO_DOWNLINK
  entry: Set all packets in `TlmPacketizer` to have a `RateLogic` of `SILENCED`. Set `downlinkMode`
   ↪ telemetry to `NO_DOWNLINK`.
```

Reference: FPP inherited transitions, FPP substates

Telemetry Groups

[TBD] on Telemetry Groups.

Communications Connections

The following diagrams highlight the routes in which uplink/downlink data (packets, data products, and events) are handled within our design:

TmtcRadioManager Connections

`TmtcRadioManager` provides the byte-stream adapter boundary between the CCSDS services and `LinuxUartDriver`. It will deliver uplink and downlink data packets to and from the rest of the flight software.

`LinuxUartDriver` will deliver uplinked bytes from the S-Band transceiver over serial to `TmtcRadioManager` through its `drvReceiveIn` input port. `TmtcRadioManager` will send downlink bytes through its `drvSendOut` output port to `LinuxUartDriver`.

Uplinked Data Connection Route

The following diagrams detail the route of downlink data through the port topology of the software subsystems.

Receiving data from Transceiver

The following diagrams outline the For uplinked data, after it has passed through the S-Band transceiver and into `TmtcRadioManager`, it will be passed to `FrameAccumulator`, which is responsible for extracting complete CCSDS TC transfer frames. `TmtcRadioManager` will transmit these received frames over its `dataOut` port into `FrameAccumulator`'s `dataIn` port.

FrameAccumulator

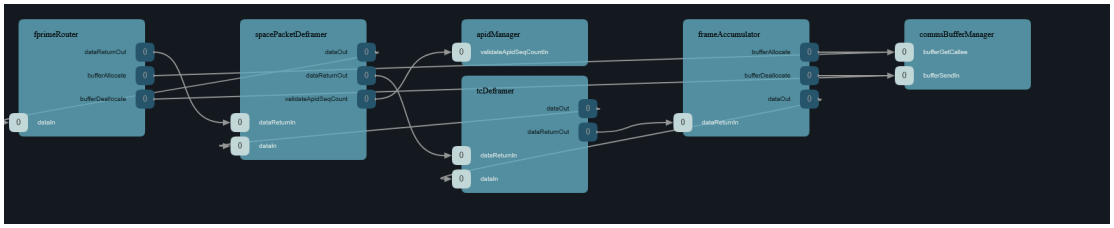


Figure 11.2: FCUplinkConnections

Once the `frameAccumulator` has received enough data to recognize a frame, it is then passed to the `Svc::Ccsds::TcDeframer` for handling. The frame is passed from the `frameAccumulator`'s `dataOut` port into the `TcDeframer`'s `dataIn` port.

Next, the `TcDeframer` will unwrap CCSDS Space Packets from the CCSDS TC frame. The unwrapped Space Packet is sent from the `dataOut` port into the `svc::Ccsds::SpacePacketDeframer`'s `dataIn` port. From here, the `SpacePacketDeframer` component extracts the uplinked data (command or file) after validating the packet. The extracted payload is sent out on the `dataOut` port and into the `Svc::FPrimeRouter`'s `dataIn` port.

The `Svc::FPrimeRouter` supports two kinds of packets: `Fw::ComPacketType::FW_PACKET_COMMAND` and `Fw::ComPacketType::FW_PACKET_FILE`. Unknown packet types are forwarded on the `unknownDataOut` port.

Command Dispatching

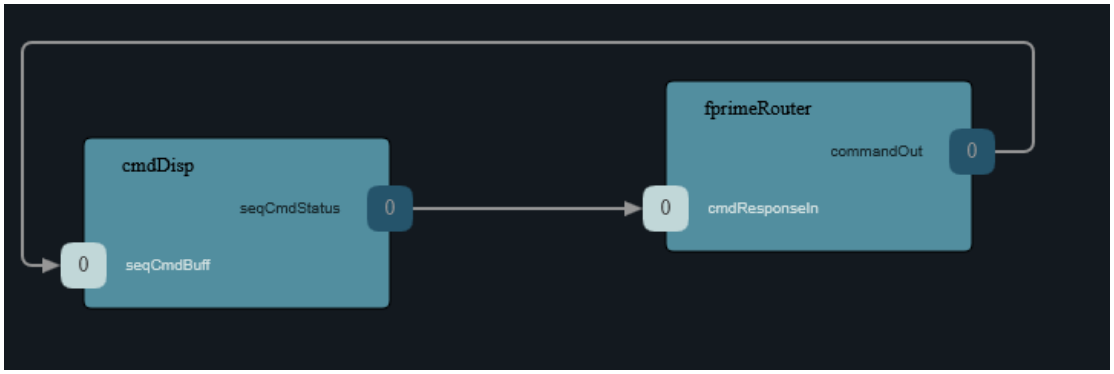


Figure 11.3: fprimeRouterToCmdDisp

Commands are sent from the `commandOut` port and to the `Svc::CmdDispatcher`'s `seqCmdBuff` input port.

Uplinked Files

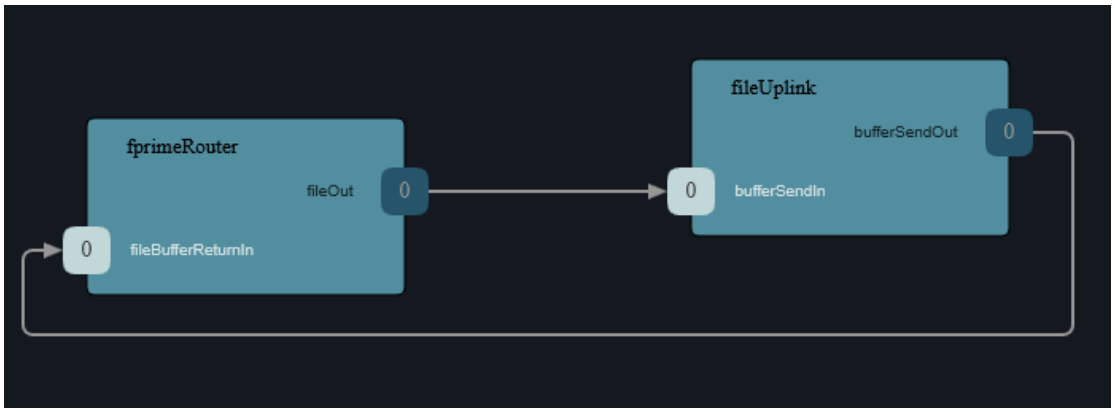


Figure 11.4: fprimeRouterToFileUplink

Uplinked file packets are sent from the `fileOut` port to the `Svc::FileUplink`'s `bufferSendIn` input port for processing.

Downlinked Data Connection Route

The following diagrams detail the route of downlink data through the port topology of the software subsystems.

Events and Telemetry Packets

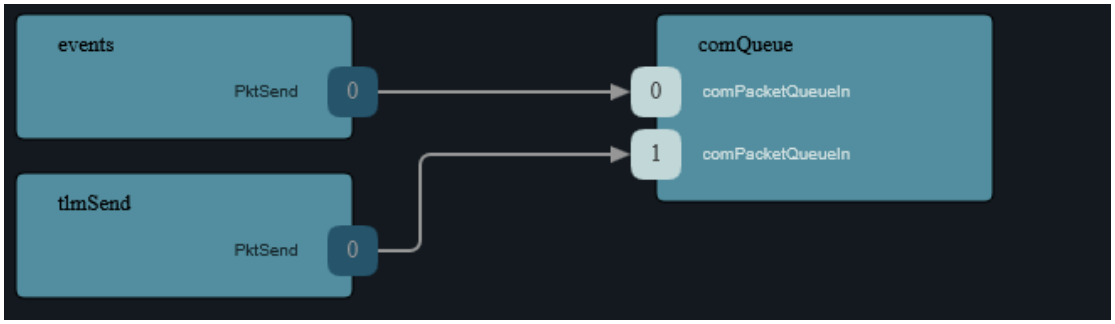


Figure 11.5: Events + TLM + ComQueue

The `Svc.ComQueue` maintains 3 different queues for downlink data; 2 queues for `Fw::ComBuffer`s (telemetry + events), 1 queue for `Fw::Buffer`s (file downlink).

Events are sent through the `Svc::EventManager`'s `PktSend` port as `ComBuffer` types into the `ComCCSDS::ComQueue`'s `comPacketQueueIn` port for processing on the internal com queue.

Telemetry packets are sent from the `Svc::TlmPacketizer`'s `PktSend` output port into the `ComCCSDS::ComQueue`'s `comPacketQueueIn` port for processing on the internal com queue.

Downlinked Files

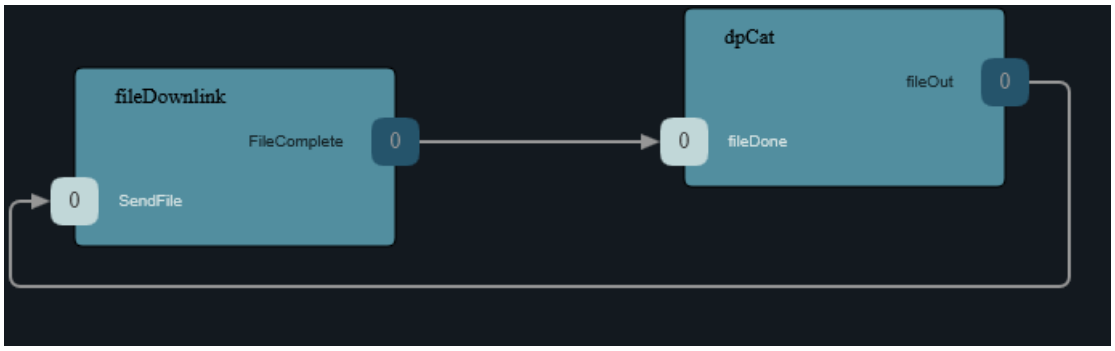


Figure 11.6: Dp Catalog to File Downlink

The `Svc.DpCatalog` component is responsible for maintaining catalogues of generated data products (files) that can be built and downlinked upon command from the operators. Downlinked files are passed from the `fileOut` output port into the `Svc.FileDownlink`'s `SendFile` input port to be enqueued for downlink.

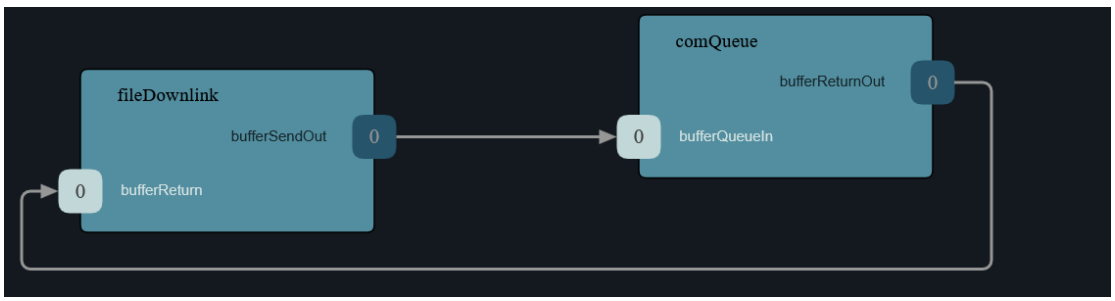


Figure 11.7: File Downlink to ComQueue

The `Svc.FileDownlink` component forwards file packets from the `bufferSendOut` port to the `Svc.ComQueue`'s `bufferQueueIn` port to append the file packet to the `ComQueue`'s internal buffer queue.

Downlink Framing

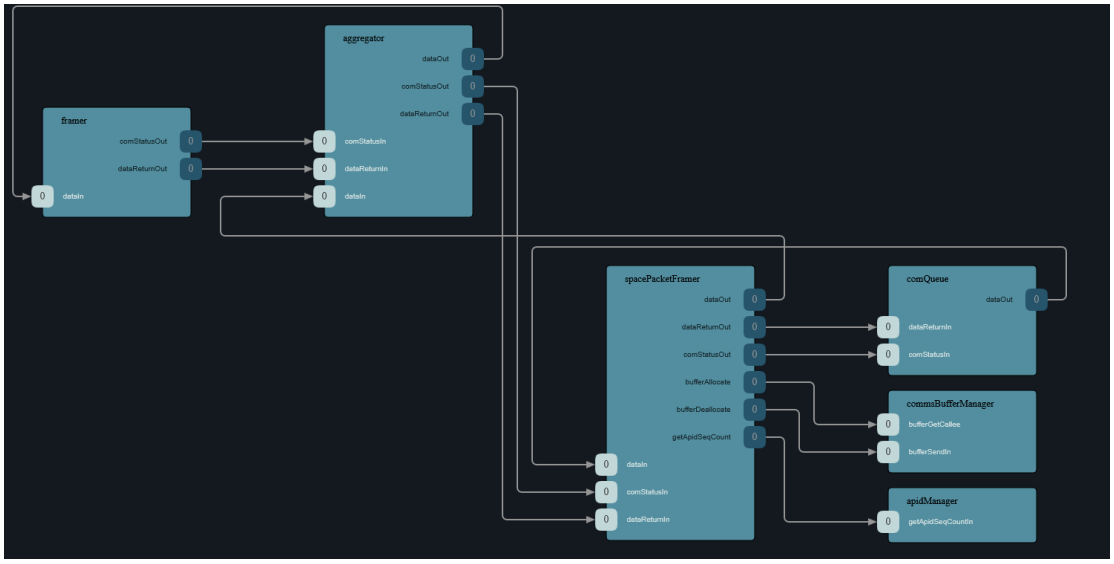


Figure 11.8: Downlink Framing

The `Svc.ComQueue` component processes `Fw::ComPacket`s and `Fw::Buffer`s to downlink telemetry/events and files from its internal queues when it is in the `READY` state (no queued buffers and is ready for sending to the communication adapter) and when the `TmtcRadioManager` indicates it is ready for another packet through the `ComQueue.comStatusIn` input port.

NOTE: The `TmtcRadioManager` must send `Fw::Success::SUCCESS` over the `comStatusOut` port to receive more data to transmit.

The data sent out from the `ComQueues dataOut` port is first sent to the `Svc.SpacePacketFramer` which wraps the downlink data in the payload of a CCSDS Space Packet. From there, the data is sent on the `Svc.SpacePacketFramer`'s `dataOut` port to the `Svc.ComAggregator`'s `dataIn` port to aggregate data for a CCSDS TM Transfer frame. Once the internal buffer is full, it is forwarded from the `dataOut` port into the `Svc.TmFramer` component's `dataIn` port.

TmtcRadioManager Downlink

After `Svc.TmFramer` wraps a CCSDS Space Packet into a CCSDS TM transfer frame, the data will be sent from its `dataOut` port into `TmtcRadioManager`'s `dataIn` port.

From there, `TmtcRadioManager` will send all downlink data through its `drvSendOut` output port into `LinuxUartDriver` for transmission to the S-band transceiver.

Communication Security

The downlink and uplink data paths will utilize the F'-provided `ComCcsdsSdls` subtopology to encrypt downlink data and decrypt uplinked data on a frame-by-frame basis. The following list clarifies the components used and their respective purposes:

- `CcsdsSdlsFramer`: Sits inbetween the `SpacePacketFramer` and `TmFramer` to add a security header and trailer to the CCSDS TM transfer frame's data field (Encryption)
- `CcsdsSdlsDeframer`: Sits inbetween the `TcDeframer` and `SpacePacketDeframer` to remove security header and trailer from the CCSDS TM transfer frame's data field (Decryption)

- `SdlsSaRouter`: Routes CCSDS SDLS encryption and decryption requests to downstream crypto components (encryptors or decryptors).

Notes

- The `ComApplication` will be a part of another subtopology, titled [TBD], that also wraps the `ComCCSDS` framing subtopology and the `TmtcRadioManager` component.
- `STANDARD_DOWNLINK` and `STORED_PLAYBACK` require `AdcsApplication` to be in `AntennaPointing` mode. `SatStateMachine` is responsible for commanding both simultaneously via the translation table. `ComApplication` does not check ADCS state directly.
- `TmtcRadioManager` interface to be defined during detailed design.

- The `TlmPacketizer` component gives the `ComApplication` capabilities to configure the rate at which certain packets are sent for the various operating modes.
- For downlinking event, there are two log files to maintain: The first is for the last [TBD] minutes of events (refreshed/overwritten every [TBD] / 2 minutes) and the other which stores the last [TBD] minutes of events after the most recent `FATAL` exception. Ground can send commands to the `Svc::FileDownlink` component to retrieve these logs files held in non-volatile memory.
- The "Health" pings will be incoming from the `Svc::Health` component which will send WARN/FATAL events if a certain number of configurable ticks have elapsed before a `pingOut` is sent from the `ComApplication` component.

12 EPSApplication

1. Overview

`EPSApplication` is the Layer 3 Active component for the EPS subtopology. It monitors battery and power-system health by consuming state data published by `MpptManager` (battery/IC state) and `CurrentSensorManager` (per-rail voltage and current) on each rate group tick, and exposes a health/status struct to `SatStateMachine` via a synchronous get port so submode decisions can be made. It also accepts the panel deployment command from ground and forwards it to `DeployPanelsManager`.

Unlike other Layer 3 components, `EPSApplication` has no internal state machine. It operates identically regardless of satellite mode.

EPS software topology:

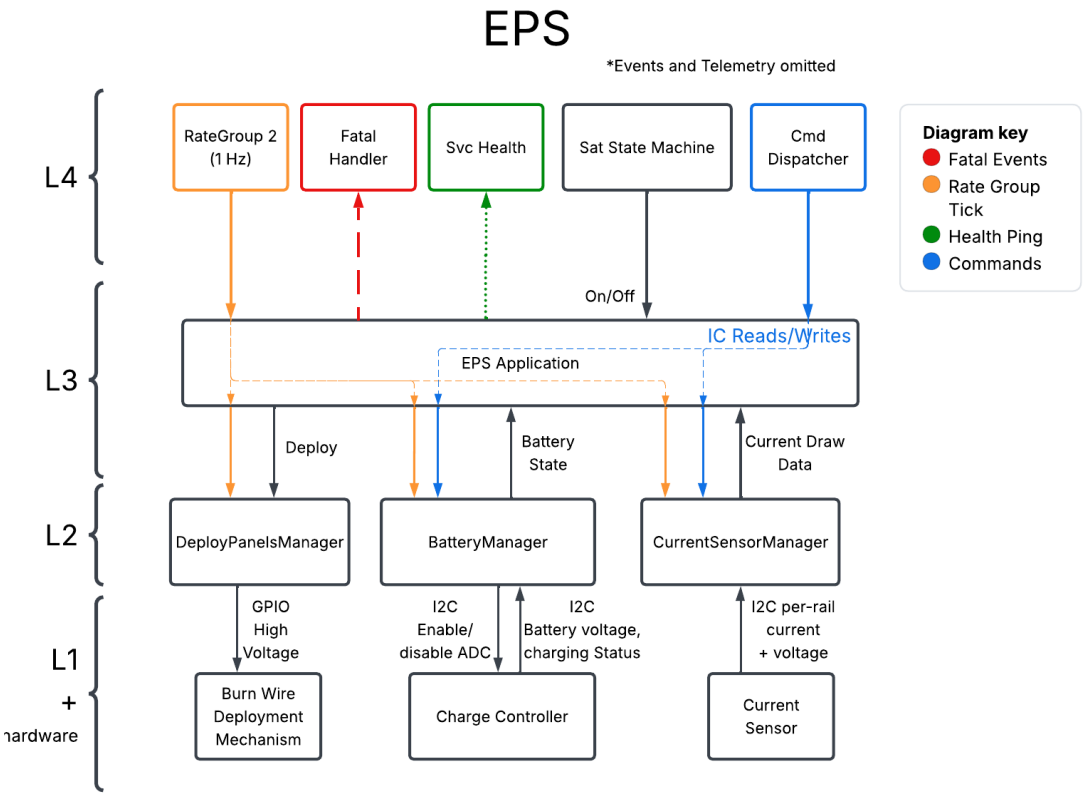


Figure 12.1: EPS application topology diagram

2. Requirements

ID	Requirement	Verification
HS2-EPS-001	EPSApplication shall expose a synchronous get port that returns the latest powerState struct when invoked by SatStateMachine.	Inspection

ID	Requirement	Verification
HS2-EPS-002	EPSApplication shall emit a WARNING_HI event when vbatt falls below POWER_THRESHOLD or CRITICAL_THRESHOLD.	Inspection
HS2-EPS-003	EPSApplication shall consume the per-rail measurements published by CurrentSensorManager and incorporate them into the cached powerState.	Inspection
HS2-EPS-004	EPSApplication shall forward DEPLOY_PANELS commands to DeployPanelsManager without refusal logic.	Inspection
HS2-EPS-005	EPSApplication shall operate continuously regardless of satellite mode.	Inspection
HS2-EPS-006	EPSApplication shall respond to health ping within the required deadline.	Inspection

3. Design

3.1 Component Type

Active component. No hierarchical state machine. `EPSApplication` operates continuously and is fully driven by rate group ticks and incoming commands with no satellite-mode-driven state transitions.

3.2 Ports

Port	Direction	Type	Purpose
`schedIn`	Input	`Svc.Sched`	Rate group tick (1 Hz)
`cmdIn`	Input	`Fw.Cmd`	Ground commands via CmdDispatcher
`cmdResponseOut`	Output	`Fw.CmdResponse`	Command completion status
`batteryStateGet`	Output	Custom `sync_input` port returning batteryState struct	Battery and IC state from MpptManager (vbatt, ibatt, vac, iac, charging status, charger/fault flags, MPPT state, temperature)
`railStateIn`	Output	Custom `sync_input` port returning railState struct	Per-rail bus voltage and current from CurrentSensorManager (12 V, 5 V, 3.3 V)
`powerStateGet`	Input	Custom `sync_input` port returning powerState struct	Synchronous get invoked by SatStateMachine; returns the latest assembled powerState (vbatt, ibatt, MPPT status, fault flags, charging status, rail voltages/currents, temperature)
`deploy`	Output	`Fw.Signal`	Trigger deployment sequence on DeployPanelsManager
`pingIn` / `pingOut`	In/Out	`Svc.Ping`	Health monitoring
`logOut`	Output	`Fw.Log`	Event logging
`prmGet`	Output	`Fw.PrmGet`	Load power threshold parameters from PrmDb

`EPSApplication` emits events (e.g. `lowBattery`, `criticalBattery`) but publishes no measurement telemetry. The periodic battery and rail measurement channels are owned and published by `MpptManager` and `CurrentSensorManager` respectively.

3.3 Commands

Mnemonic	Args	Description
`DEPLOY_PANELS`	-	Trigger panel deployment sequence via DeployPanelsManager

4. Operational Behavior

`EPSApplication` does not use a hierarchical state machine. Each 1 Hz rate group tick it reads the latest battery state received from `MpptManager` and the latest rail state received from `CurrentSensorManager`, runs protection threshold checks, emits any required fault events, and caches the assembled health struct as the current `powerState`. `SatStateMachine` retrieves that struct on demand by invoking the `powerStateGet` synchronous get port. The `DEPLOY_PANELS` handler is a thin forwarder that validates the request, calls the `deploy` port, and returns a command response.

Rate group tick flow:

```
schedIn fires (1 Hz)
  → read latest batteryState from MpptManager
  → read latest railState from CurrentSensorManager
  → check vbatt against POWER_THRESHOLD parameter
      if below WARNING threshold → log WARNING_HI (LOW_BATTERY)
      if below CRITICAL threshold → log WARNING_HI (CRITICAL_BATTERY)
  → assemble and cache powerState struct (battery + rail data)
```

Synchronous get flow (`powerStateGet`):

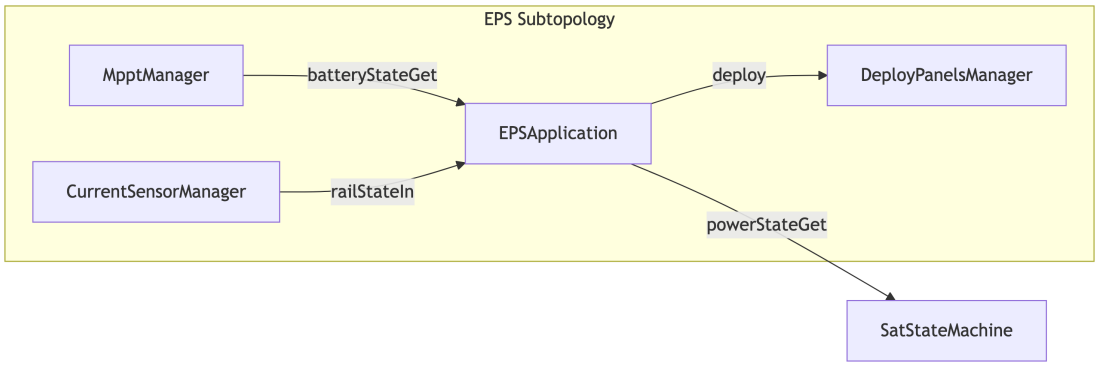
```
SatStateMachine invokes powerStateGet (caller's thread, 1 Hz)
  → return cached powerState struct
```

Command flow (`DEPLOY_PANELS`):

```
cmdIn DEPLOY_PANELS
  → call deploy port to DeployPanelsManager
  → emit activity event
  → send cmdResponse OK
```

5. Notes

Subtopology wiring:



- `EPSApplication` does not autonomously enable or disable MPPT or charging, and it no longer forwards register writes. All BQ25756 register changes are commanded directly on `MpptManager` from ground. `EPSApplication` may in the future autonomously adjust charging thresholds based on received battery and rail state. This could require an internal state machine and is TBD pending further design.
- `EPSApplication` forwards `DEPLOY_PANELS` unconditionally to `DeployPanelsManager`. Deployment state tracking and re-attempt behavior are owned by `DeployPanelsManager`'s state machine.
- `batteryStateIn` port type is a custom struct carrying all BQ25756 measurement and status data plus charger/fault flags; final type to be resolved during detailed design. Consider splitting into a measurements port and a flags port if the struct becomes unwieldy.
- `railStateIn` port type is a custom struct carrying the three rails' bus voltage and current, published by `CurrentSensorManager`. `EPSApplication` currently caches it into `powerState`; more complex per-rail fault logic is deferred.

- `powerStateGet` is a synchronous get port (`sync_input` on `EPSApplication`), called from `SatStateMachine`'s caller thread on its 1 Hz tick. The returned struct is the cached value last assembled by the EPS rate group tick; no recomputation occurs inside the get handler. Struct type is custom and must carry at minimum vbatt, ibatt, MPPT status, fault flags, and charging status for `SatStateMachine` submode evaluation; rail voltage/current are included for completeness.
- `BQ25756Reg` enum definitions and register semantics are owned alongside `MpptManager` and produced during detailed design; `EPSApplication` no longer references them.
- `MpptManager`, `CurrentSensorManager`, `WatchdogPinger`, and `DeployPanelsManager` are all instantiated within the EPS subtopology. `EPSApplication` is health-monitored; the hardware managers are not.
- Power threshold parameters (`POWER_THRESHOLD`, `CRITICAL_THRESHOLD`) persisted via `PrmDb`. Specific threshold values and actions TBD pending battery characterization testing.

13 ScienceApplication

ScienceApplication is the Layer 3 active component for the Science subtopology. On a schedule, it reads a manifest of imaging opportunities logged on the external disk, finds the first one that has every image type a ground-configured algorithm chain needs, and runs that chain against it.

Payload software topology:

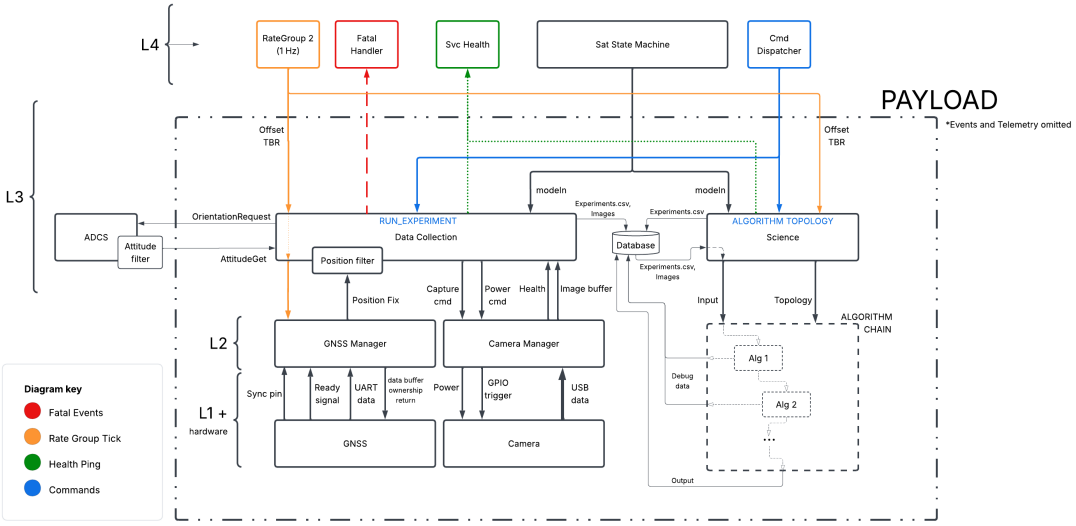


Figure 13.1: Payload software topology diagram

The component is driven by two synchronous input ports:

- `schedIn` (`Svc.Sched`) — a rate-group tick. All manifest scanning and processing happens here.
- `modeIn` (`Sat.ScienceModePort`, carrying `Science.Mode`) — mode commands from `SatStateMachine`. `Off` holds the component idle; `ProcessImages` starts the per-tick scanning loop.

File system
The partition holds two things: a manifest, and one subdirectory per `Science.ImageType` integer value. `<imagePartitionDir>/experiments.csv` logs imaging opportunities.

Each experiment opportunity has:

- `timeRequested` as `YYYY:MM:DD:HH:MM:SS`,
- `timeDataCollected` as `YYYY:MM:DD:HH:MM:SS`,
- `timeScienceStarted` as `YYYY:MM:DD:HH:MM:SS`,
- `timeScienceFinished` as `YYYY:MM:DD:HH:MM:SS`,
- `positionKnown` as `bool`,
- `position` as `x:y:z`
- `attitude` as `x:y:z:w` (quaternion),
- `availableImageTypes` a `U16` bitmask representing `Science.ImageType`
- `experimentID` as a `U16`

`<imagePartitionDir>/1/` holds `STARS` images, `<imagePartitionDir>/2/` holds `HORIZON` images, etc.
`fileName` is the experiment ID

Requirements

ID	Requirement	Verification
HS2-SIA-001	ScienceApplication shall process at most one `experiments.csv` opportunity per `schedIn` tick while in `ProcessImages` mode	Unit test
HS2-SIA-002	Ground shall be able to set and clear individual algorithm-topology slots (0-9) via `SET_ALGORITHM/CLEAR_ALGORITHM`, rejecting an out-of-range index with `VALIDATION_ERROR`, and clear the whole topology at once via `CLEAR_SCIENCE_TOPOLOGY`	Unit test
HS2-SIA-003	ScienceApplication shall find the first `experiments.csv` line whose `availableImageTypes` bitmask covers every configured algorithm's `inputImage`, and run the configured chain, in slot order, against it	Unit test
HS2-SIA-004	ScienceApplication shall preserve a copy of an algorithm's image content in `processed/` *before* running it, since the algorithm may overwrite that same path in place	Unit test
HS2-SIA-005	ScienceApplication shall log/telemeter each processed image's outcome (path, algorithm name, flagged) as part of the downlinked record	Unit test
HS2-SIA-006	ScienceApplication shall request compression for images the algorithm flags (external-process exit code 2)	Unit test
HS2-SIA-007	ScienceApplication shall emit `WARNING_HI` and leave the manifest line in `experiments.csv` if any algorithm in the chain fails	Unit test
HS2-SIA-008	ScienceApplication shall emit `WARNING_HI` and leave the file in place if it cannot preserve the pre-run image content in `processed/`, retrying on a later tick	Unit test
HS2-SIA-009	ScienceApplication shall switch between `Off` and `ProcessImages` on command from `SatStateMachine`, logging `StateChanged` on every mode command	Unit test
HS2-SIA-010	ScienceApplication shall respond to `pingIn` immediately on `pingOut` with the same key	Unit test
HS2-SIA-011	Once every configured algorithm has succeeded against an experiment, ScienceApplication shall move that experiment's manifest line from `experiments.csv` to `completeExperiments.csv`, tagged with the ordered list of algorithms that ran	Unit test

Design

Ports

Port	Kind	Direction	Type	Usage
`modeIn`	sync	input	`Sat.ScienceModePort`	Mode command from `SatStateMachine`.
`schedIn`	sync	input	`Svc.Sched`	Rate-group tick; sends `tick` to the state machine on every call.
`compressRequestOut`	—	output	`Science.CompressRequest`	Requests compression of a flagged image, by its (single, shared) image path.
`pingIn` / `pingOut`	sync / —	in / out	`Svc.Ping`	Health monitoring; every `pingIn` is echoed immediately on `pingOut`.

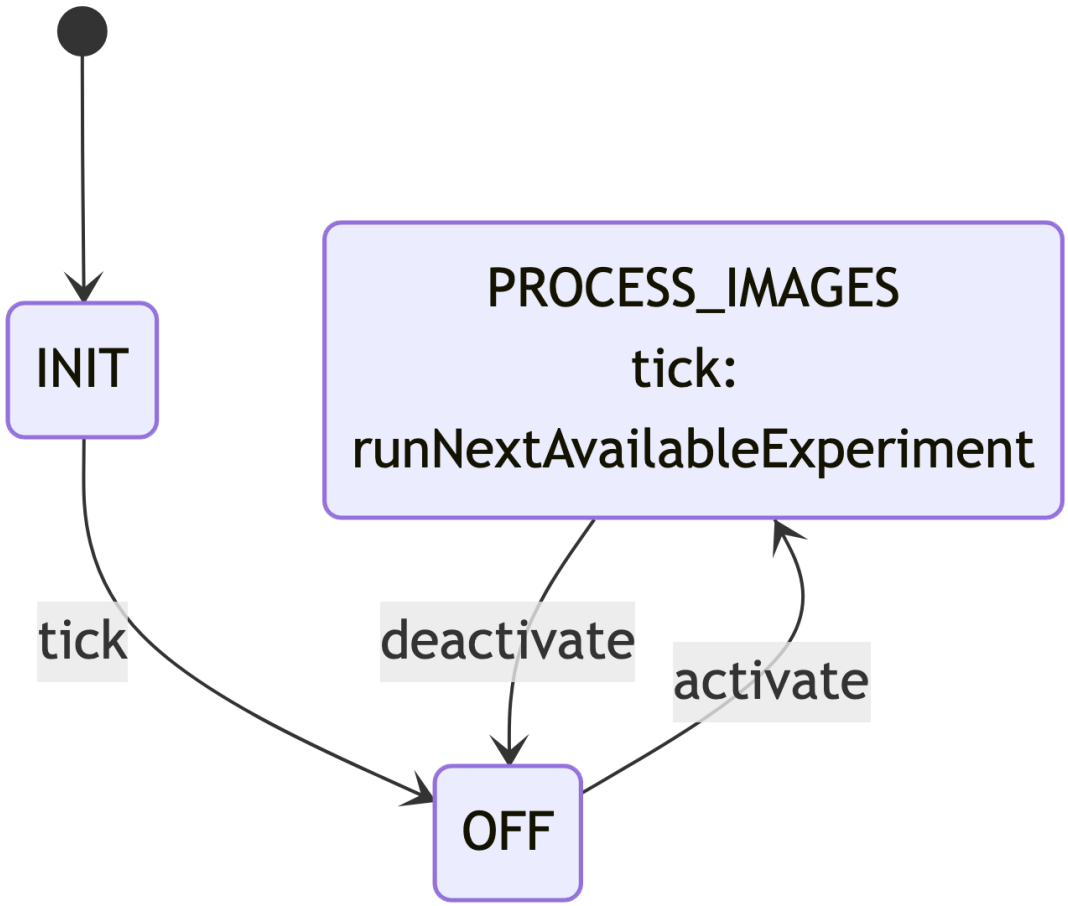
Port	Kind	Direction	Type	Usage
`timeCaller`, `Fw.Command`, `Fw.Event`, `Fw.Channel`	standard AC ports	—	—	Boilerplate command/event/telemetry/time wiring.

Commands

Name	Arguments	Effect
`SET_ALGORITHM`	index (U8), algorithm (Science.Algorithm)	Sets topology slot `index` to `algorithm`
`SET_ALGORITHM_PRESET`	index (U8), preset (Science.AlgorithmPreset), inputImage (Science.ImageType), camera (Science.Camera)	Sets topology slot `index` to a named predefined algorithm
`CLEAR_ALGORITHM`	index (U8)	Resets topology slot `index` to an unconfigured `Science.Algorithm`.
`CLEAR_SCIENCE_TOPOLOGY`	—	Resets every one of the 10 topology slots to a default `Science.Algorithm`

State Machine

`sciAppStateMachine` (`Science_ScienceApplicationStateMachine_t`, defined in `ScienceApplicationStateMachine.fpp`) tracks operating mode.



State	Meaning	On `tick`
`INIT`	Idle until the first `schedIn` tick, which transitions out immediately without touching any output port.	(transitions to `OFF`)
`OFF`	Idle: `experiments.csv` is never read, no images processed.	ignored
`PROCESS_IMAGES`	Steady state: at most one experiment is found and its full algorithm chain processed per tick.	`runNextAvailableExperiment`

14 ThermalApplication

ThermalApplication is the Layer 3 active component for the Thermal subtopology. On a 1 Hz rate-group tick, it reads all onboard temperature sensors (via `TemperatureSensorManager`), runs a PID control loop against a commanded setpoint, and commands the resulting duty cycle to the heater. Svc::FatalHandler responds to "overheating" warnings emitted by this application to shut off hot hardware.

The component is driven by two synchronous input ports:

- `schedIn` (`Svc.Sched`) — a 1 Hz rate-group tick. All sensor polling and PID computation happens here.
- `modeIn` (`Sat.ThermalModePort`, carrying `Thermals.Mode`) — mode commands from `SatStateMachine`.

Thermal software topology:

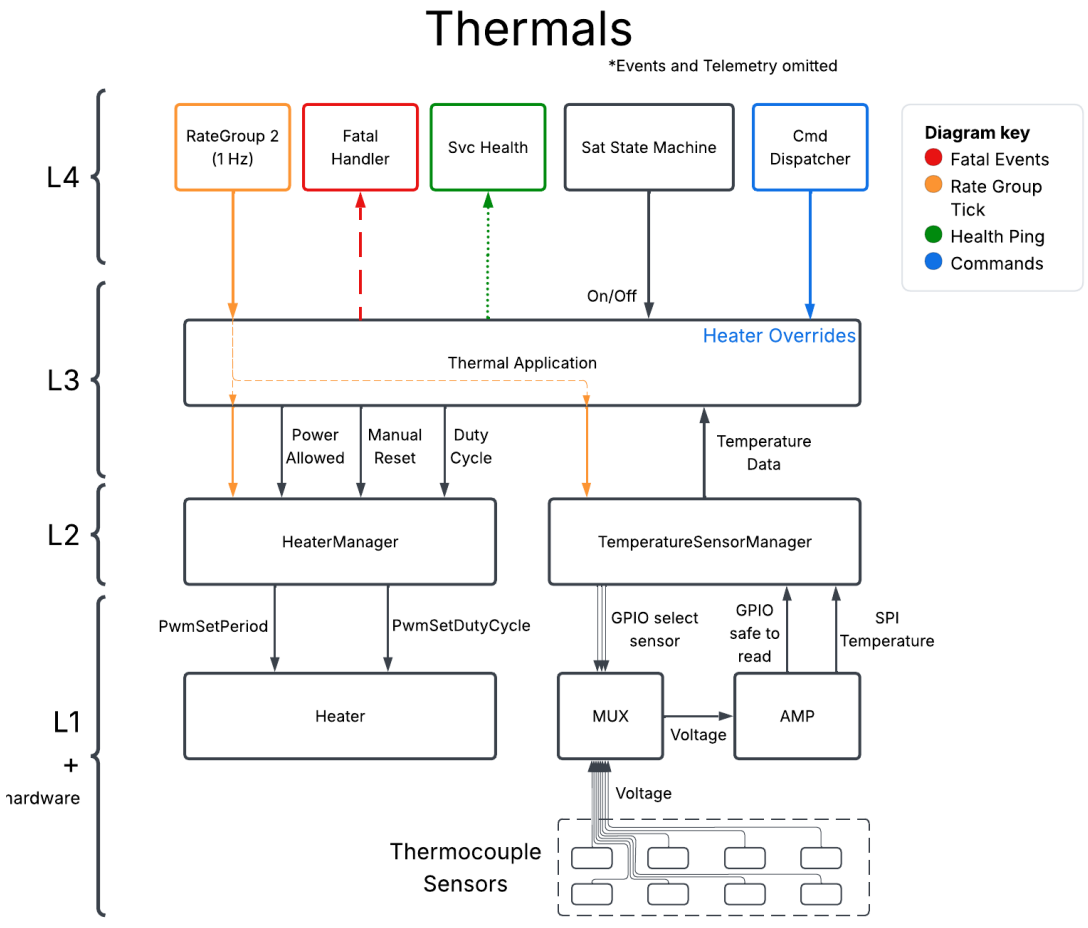


Figure 14.1: Thermal application topology diagram

Requirements

ID	Requirement	Verification
HS2-THA-001	ThermalApplication shall hold the commanded heater duty cycle at 0% while in `NO_HEATING`, regardless of any active `SET_HEATER_OVERRIDE`	Unit test
HS2-THA-002	While in `ACTIVE_HEATING`, ThermalApplication shall run a PID control loop each tick using the difference between the setpoint and the average of valid sensor readings, combine proportional, integral, and derivative terms, and command the result to HeaterManager after clamping it to the 0-100 percent range.	Unit test
HS2-THA-003	On any invalid sensor reading, ThermalApplication shall log `SensorInvalid` for that sensor. If fewer than `MIN_VALID_SENSORS` readings remain valid, it shall additionally log `InsufficientValidSensors`, reset the PID integrator and derivative history, and command 0% duty (subject to override) for that tick.	Unit test
HS2-THA-004	ThermalApplication shall publish every sensor's raw reading and validity flag to the `TempC` telemetry channel each `ACTIVE_HEATING` tick	Unit test
HS2-THA-005	ThermalApplication shall accept `SET_HEATER_OVERRIDE` to force a fixed, clamped duty cycle regardless of PID output or sensor validity, and `CLEAR_HEATER_OVERRIDE` to resume normal PID-driven control; the PID integral term shall not accumulate for any tick during which an override is engaged	Unit test
HS2-THA-006	ThermalApplication shall switch between `NO_HEATING` and `ActiveHeating` on command from `SatStateMachine`, logging `StateChanged` on every mode command	Unit test
HS2-THA-007	ThermalApplication shall accept parameter inputs `SENSOR_COLD_THRESH` and `SENSOR_HOT_THRESH` for every sensor to specify acceptable temp ranges	Unit test
HS2-THA-008	ThermalApplication shall emit warning HI specifying which unit is overheating on any sensor reading over its designated threshold	Unit test
HS2-THA-009	ThermalApplication shall respond to `pingIn` immediately on `pingOut` with the same key	Unit test

Design

Ports

Port	Kind	Direction	Type	Usage
`modeIn`	sync	input	`Sat.ThermalModePort`	Mode command from `SatStateMachine`.
`schedIn`	sync	input	`Svc.Sched`	1 Hz rate-group tick; sends `tick` to the state machine on every call.
`tempReadOut`	sync	output	`Thermals.TempRead`	Synchronous read of `TemperatureSensorManager`'s cached readings (array of `{temperatureC, valid}`, one per sensor).
`heaterCmdOut`	sync	output	`Thermals.HeaterCmd`	Duty-cycle command (percent) to `HeaterManager`.

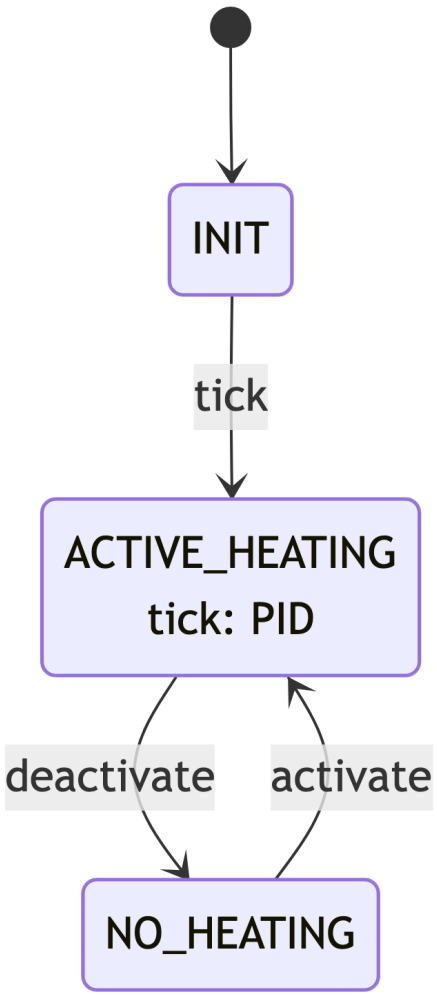
Port	Kind	Direction	Type	Usage

Commands

Name	Arguments	Effect
`SET_HEATER_OVERRIDE`	dutyPercent (F32)	Clamps `dutyPercent` to the 0-100 percent range, engages the override (stored, applied by every subsequent `commandHeaterDuty` call regardless of state), logs `OverrideEngaged` with the clamped value, responds `OK` unconditionally.
`CLEAR_HEATER_OVERRIDE`	—	Disengages the override; subsequent duty commands reflect the PID loop (or 0%, in `NO_HEATING`) again. Logs `OverrideCleared`, responds `OK` unconditionally.

State Machine

`thrmAppStateMachine` (`Thermals_ThermalApplicationStateMachine_t`, defined in `ThermalApplicationStateMachine.fpp`) switches between idle and the per-tick control loop.



Parameters

Name	Type	Default	Purpose
`Kp`	`F32`	`5.0`	Proportional gain for the heater control loop.
`Ki`	`F32`	`0.1`	Integral gain for the heater control loop.
`Kd`	`F32`	`0.0`	Derivative gain for the heater control loop. Defaults to `0.0` (a no-op) — not tuned against real hardware yet.
`SetpointC`	`F32`	`20.0`	Target temperature setpoint, in Celsius, that the control loop drives the sensor average toward.
`SENSOR_COLD_THRESH` (one per sensor)	`F32`	-	Per sensor minimum used to decide whether to activate heaters.
`SENSOR_HOT_THRESH` (one per sensor)	`F32`	-	Per sensor maximum used to emit a fatal warning to shut off nearby components.

Part III

Level 2 - Hardware Managers

15 ADCS Hardware Managers

The ADCS hardware-manager layer owns sensor and actuator protocols for attitude determination and control. Each manager follows the shared initialization and recovery pattern while `AdcsApplication` selects mission behavior and control targets.

The following component designs cover the star tracker, IMU, sun sensors, and magnetorquer PWM control.

15.1 ImmuManager

ImmuManager is a Layer 2 Queued worker component in the ADCS subtopology. It owns the VN-100 IMU/AHRS sensor, bringing it up over a serial link through an internal state machine and serving cached accel/gyro/magnetometer/temperature readings to the rest of ADCS.

Introduction

The component is driven by two asynchronous input ports:

- `run` (`Svc.Sched`) — rate-group tick; every call sends `tick` to the state machine.
- `Reset` (`Fw.Signal`) — zeroes `tickCount` and forces the state back to `RESET`.

The sensor itself is a VectorNav VN-100, accessed through the vendor `VN::Sensor` C++ SDK over a serial (RS-232/UART) connection. Rather than calling `VN::Sensor` directly, ImmuManager talks to an `IImmuSensor` interface: `VnSensorAdapter` is the production implementation (it owns the real `VN::Sensor` plus its 64 KiB main / 2 KiB FB byte buffers and forwards each call), while unit tests inject a gmock `IImmuSensor` through the test-only constructor. `configureDataRequest` asks the owning application for the connection parameters to use.

Requirements

Name	Description	Validation
IMU-001	While entering `CONFIGURE`, the component shall request connection parameters via `configureDataRequest` and connect to the VN-100 at the requested port / baud rate.	Unit Test
IMU-002	While in `CONFIGURE`, the component shall set the sensor's `AsyncOutputFreq` register to the faster of the requested accel/gyro and magnetometer data rates.	Unit Test
IMU-003	While in `WAIT_STARTUP`, the component shall wait until `WAIT_SECONDS` seconds have elapsed before advancing to `ACTIVE`.	Unit Test
IMU-004	While in `ACTIVE`, the component shall read the VN-100 IMU register every tick and refresh its cached accel/gyro/temperature at `ACCEL_GYRO_DATA_RATE_HZ` and its cached magnetic field at `B_DATA_RATE_HZ`.	Unit Test
IMU-005	The component shall serve the most recently cached reading, with its measurement timestamp, on `getIMUData`, and `getMagnetometer`.	Unit Test
IMU-006	The component shall forward `magnetorquerEnabled` requests to the sensor.	Unit Test
IMU-008	On any VN-100 register read/write or connect failure, the component shall log `immuError` with the SDK error code and drive the state machine back to `RESET` via `error`.	Unit Test

Design

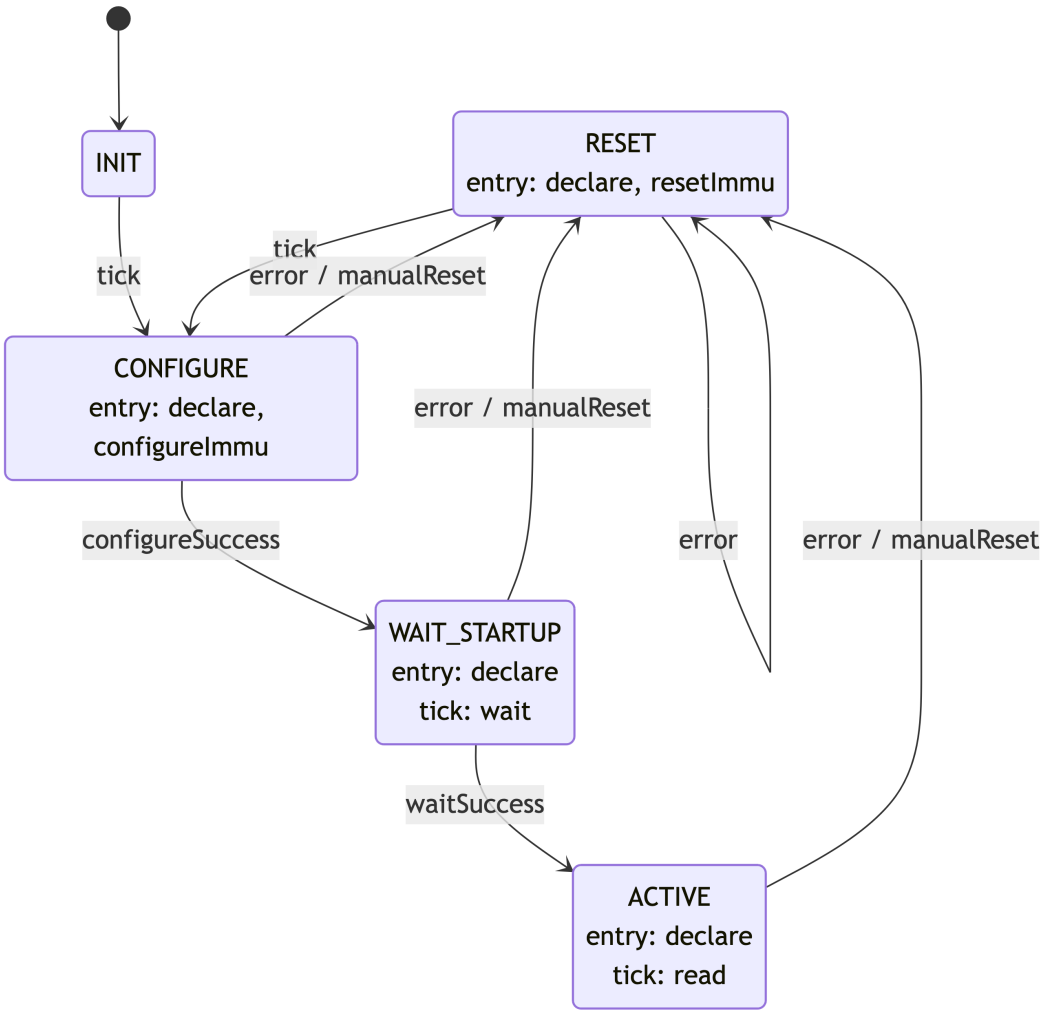
Ports

Port	Kind	Direction	Type	Usage
`run`	async	input	`Svc.Sched`	Rate-group tick; sends `tick` to the state machine on every call.

Port	Kind	Direction	Type	Usage
`Reset`	async	input	`Fw.Signal`	Zeroes `tickCount` and sends `manualReset` to the state machine.
`getIMUData`	sync	input	`imuDataRead_p` → `Adcs.RawIMUData`	Returns cached accel, gyro, and temperature with the accel/gyro measurement timestamp.
`getMagnetometer`	sync	input	`bDataRead_p` → `Adcs.RawBData`	Returns cached magnetic field with its measurement timestamp.
`magnetorquerEnabled`	sync	input	`mtToggle_p` with is_locked boolean	Latches `magnetometerLocked` and forwards `Present` / `NotPresent` to the sensor's `knownMagneticDisturbance`.
`configureDataRequest`	—	output	`immuConfigurationParameters_p` → `Adcs.ConfigureParameters`	Asks the owning application for port name, data rates, and startup wait time; called once per `CONFIGURE` entry.

State Machine

`immuStateMachine` (`Adcs_IMMUStateMachine_t`, defined in `IMMUStateMachine.fpp`) owns bring-up and fault recovery. Its telemetered state is the `Adcs.ImmuState` enum, mirrored into the `SMstate` member by the `declare` action.



State	Meaning	Entry actions	On `tick`
`INIT`	Idle until the first `run` tick.	—	proceed to `CONFIGURE`
`RESET`	Disconnects from the sensor, then retries bring-up.	`declare`, `resetImmu`	proceed to `CONFIGURE` (paces reconnect retries at the tick rate)
`CONFIGURE`	Requests connection parameters, connects, and configures the sensor's output-frequency register.	`declare`, `configureImmu`	ignored
`WAIT_STARTUP`	Holds for `WAIT_SECONDS` seconds to let the VN-100 finish its own boot sequence.	`declare`	`wait`
`ACTIVE`	Steady state: reads the IMU register every tick, downsampling the cached values to the configured per-signal rates.	`declare`	`read`

Configuration parameters

These are not F´ parameters — they arrive at runtime from the owning application as an `Adcs.ConfigureParameters` struct returned by `configureDataRequest`, and are cached in the `config` member on each `CONFIGURE` entry.

Field	Type	Meaning
`PORT_NAME`	`string`	Serial port to connect the VN-100 on.
`ACCEL_GYRO_DATA_RATE_HZ`	`U16`	Target rate for cached accel/gyro/temperature updates.
`B_DATA_RATE_HZ`	`U16`	Target rate for cached magnetic-field updates; **also** passed to `VN::Sensor::BaudRate(...)` as the connection baud rate (see Open Items).
`WAIT_SECONDS`	`U8`	Post-configure hold time before `ACTIVE`.

`immuPortHz` (the `run` port's tick rate) is currently hardcoded to `16` in the constructor, pending a shared constants file.

15.2 SunSensorManager

1. Overview

`SunSensorManager` is the Layer 2 hardware manager for the sun sensor array within the ADCS subtopology. It owns all bus communication with the sun sensors, confirming the ADC is alive and reading and calibrating each sensor channel, and publishes the calibrated intensities to `AdcsApplication` each rate-group tick.

`SunSensorManager` is a **Queued** component driven by `RateGroup1` (10 Hz). It has no satellite mode awareness; it runs its startup and read loop unconditionally once initialized. `AdcsApplication` (Layer 3) is responsible for computing the sun vector from the calibrated intensities, interpreting it, acting on failures, and, if necessary, commanding a power cut to the sensor array via EPS.

The HS2 sun sensor array consists of six photodiode sensors mounted on the solar panels read through an **MCP3208 SPI ADC**. The MCP3208 is a 12-bit converter with no writable configuration registers of any kind (no reset, no enable/power, no resolution control; see datasheet DS21298E §5.0). Its entire interface is a single full-duplex SPI transaction: clock in a start bit + channel-select bits, clock back that channel's 12-bit conversion result in the same exchange. All bus access goes through the Layer 1 `LinuxSpiDriver`.

2. Requirements

ID	Requirement	Verification
HS2-SSM-001	SunSensorManager shall confirm the sun sensor ADC is present and responding before proceeding out of RESET	Inspection
HS2-SSM-002	SunSensorManager shall wait for the reset stabilization period before proceeding to RUN	Inspection
HS2-SSM-003	SunSensorManager shall pull the ADC's chip select pin low via GPIO before each SPI transaction (RESET ping and RUN channel reads) and release it high immediately afterward	Inspection
HS2-SSM-004	SunSensorManager shall read all six sun sensor channels each rate-group tick while in RUN	Inspection
HS2-SSM-005	SunSensorManager shall apply the calibration model (per-channel scale and offset) to each raw channel reading to produce a calibrated intensity value	Inspection
HS2-SSM-006	SunSensorManager shall publish the six calibrated channel intensities to AdcsApplication after each successful read cycle	Inspection
HS2-SSM-007	SunSensorManager shall log a WARNING_HI event and transition to RESET on any SPI read or write failure	Inspection
HS2-SSM-008	SunSensorManager shall track consecutive failure count and report it as a telemetry channel	Inspection
HS2-SSM-009	SunSensorManager shall apply updated F' parameter values by reloading them from PrmDb while remaining in RUN, without a full reset	Inspection
HS2-SSM-010	SunSensorManager shall not perform any bus operations while in WAIT_RESET	Inspection
HS2-SSM-011	SunSensorManager shall expose the most recently cached calibrated channel intensities via a synchronous getter port, so a consumer can obtain the current values on demand rather than waiting for the next RUN publish cycle	Inspection
HS2-SSM-012	SunSensorManager shall transition to OFF from any state on receipt of an OFF command via controlIn	Inspection
HS2-SSM-013	SunSensorManager shall perform no bus operations while in OFF	Inspection
HS2-SSM-014	SunSensorManager shall re-enter RESET on receipt of an ON command via controlIn while in OFF	Inspection
HS2-SSM-015	SunSensorManager OFF entry hardware shutdown sequence is TBD pending ADC hardware selection	Deferred

3. Design

3.1 Component Type

Queued component with internal flat F' state machine (`Fw::Sm`). Has a message queue but no dedicated thread; it executes on the `RateGroup1` caller thread each 10 Hz tick.

3.2 Parameters

`SunSensorManager` is not satellite-mode-aware and receives no mode commands. Configuration is entirely parameter-driven. Parameters are loaded from `Svc::PrmDb` during the `WAIT_RESET`-to-`RUN` transition. A parameter update at runtime reloads all parameters from `PrmDb` in place while remaining in `RUN`.

Parameter	Type	Description
`CALIBRATION_SCALE_0…CALIBRATION_SCALE_5`	`F32` (×6)	Per-channel sensitivity scale factors
`CALIBRATION_OFFSET_0…CALIBRATION_OFFSET_5`	`F32` (×6)	Per-channel dark-current offset corrections
`RESET_WAIT_TICKS`	`U32`	Number of rate-group ticks to wait after reset before proceeding

3.3 Ports

Port	Direction	Type	Purpose
`schedIn`	Input	`Svc.Sched`	10 Hz rate group tick that drives the SM step
`getSunIntensities`	Input (sync)	Custom port (`intensityGetter_p`)	Returns the most recently cached calibrated channel intensities on demand, independent of the tick cycle
`csOut`	Output	`Drv.GpioWrite`	Pulls the ADC's chip select pin low before each SPI transaction; released high immediately after
`spiWriteRead`	Output	`Drv.SpiWriteRead`	SPI transaction with the ADC, used for the RESET ping and channel reads. One transaction per channel.
`sunIntensitiesOut`	Output	`Adcs.SunIntensityPort`	Publish the six calibrated channel intensities + timestamp directly to `AdcsApplication`'s `sunIntensitiesIn` port; `AdcsApplication`'s internal `SunVectorEstimator` converts them into a body-frame sun vector (formerly `AttitudeFilter`'s role, which never actually specified this conversion)
`controlIn`	Input (async)	`Adcs.ManagerControlPort`	ON/OFF command from AdcsApplication
`prmGet`	Output	`Fw.PrmGet`	Load parameters from PrmDb during the WAIT_RESET-to-RUN transition, and on parameter updates thereafter
`logOut`	Output	`Fw.Log`	Event logging (state transitions, errors)
`tlmOut`	Output	`Fw.Tlm`	Telemetry (SM state, consecutive failure count, raw and calibrated channel intensities)

3.4 Commands

`SunSensorManager` accepts no ground commands. It is not health-monitored. All recovery is handled autonomously by the self-healing SM or escalated via telemetry to `AdcsApplication`.

4. State Machine

SunSensorManager uses a single flat F' state machine following the fprime-community/fprime-sensors ImuManager reference pattern. All states are peers; there is no nesting.

```
OFF
  entry: zero all channel intensity buffers [further shutdown protocol TBD - see HS2-SSM-015]
  on tick: no action (no bus operations)
  on controlIn(ON) → RESET

RESET
  entry: zero all channel intensity buffers
         reset wait-tick counter
         pull csOut low, ping channel 0 over SPI to confirm the ADC is
         present and responding, release csOut high
  on ping success → WAIT_RESET
  on ping failure → log WARNING_HI, increment failure count, retry

WAIT_RESET
  on tick: increment wait-tick counter
           if counter >= RESET_WAIT_TICKS:
             load CALIBRATION_SCALE_0-5, CALIBRATION_OFFSET_0-5 from PrmDb
             → RUN
           (no bus operations performed in this state)

RUN
  on tick: for each of the 6 channels:
             pull csOut low, one SPI transaction (clock out start +
             channel-select bits, clock in the 12-bit conversion result),
             release csOut high
           if any spiWriteRead error or lost conversion framing:
             log WARNING_HI (throttled), increment failure count → RESET
           if all reads OK:
             apply CALIBRATION_SCALE/OFFSET to each channel
             publish sunIntensitiesOut (6 calibrated intensities, timestamp)
             reset consecutive-failure counter
  on reconfigure signal: reload CALIBRATION_SCALE_0-5, CALIBRATION_OFFSET_0-5
                         from PrmDb in place (no state transition)

# Global signals - reachable from any state:
on error signal       → RESET   # existing - self-healing fallback
on controlIn(OFF)     → OFF     # new - immediate from any state, including mid-startup
on controlIn(ON)      → RESET   # new - only meaningful from OFF; no-op from all other states
```

Error recovery: RESET retries itself on a ping failure; RUN returns to RESET on a read error.

OFF power control: controlIn(OFF) enters OFF from RESET, WAIT_RESET, or RUN.

Powering back on: OFF --> RESET on controlIn(ON), from any state.

Error self-healing: Any SPI error or ADC framing loss from RESET or RUN emits a throttled WARNING_HI event, increments the consecutiveFailures telemetry channel, and re-enters RESET. The SM retries the full startup sequence automatically.

Repeated failure escalation: consecutiveFailures is only cleared on a fully successful RUN read and publish cycle, so entering RESET again does not reset it. SunSensorManager does not cut power to itself. If consecutiveFailures exceeds a threshold observable by AdcsApplication via telemetry, AdcsApplication is responsible for commanding an EPS power rail cut. SunSensorManager continues attempting self-healing until then.

Parameter reconfiguration: When parameterUpdated() is called while SunSensorManager is in RUN, it reloads CALIBRATION_SCALE_0-5 and CALIBRATION_OFFSET_0-5 from PrmDb in place. While in RESET or WAIT_RESET, the signal is ignored, since parameters are freshly loaded on the next WAIT_RESET-to-RUN transition.

Reference: fprime-community/fprime-sensors/ImuManager, FPP flat state machines, MCP3204/3208 datasheet DS21298E §3.7 (CS/SHDN behavior) and §5.0 (transaction framing)

5. Notes

- `SunSensorManager` is instantiated inside the ADCS subtopology. Its `spiWriteRead` port connects to a `LinuxSpiDriver` instance and its `csOut` port connects to a `LinuxGpioDriver` instance, both at the top-level topology.
- `sunIntensitiesOut` connects directly to `AdcsApplication`'s `sunIntensitiesIn` port. `AdcsApplication`'s internal `SunVectorEstimator` (see `AdcsApplication.md` §3.4) converts the six intensities into a body-frame sun vector; `AttitudeFilter` (which previously sat between them) no longer exists as a separate component.
- `SunSensorManager` is **excluded from health monitoring** (`Svc::Health`). Only `AdcsApplication` is health-checked.
- The `Adcs.SunIntensityPort` type (carrying the six calibrated channel intensities and a timestamp) is defined in the ADCS module and shared with `AdcsApplication`.
- Deferred: exact `consecutiveFailures` threshold that triggers `AdcsApplication` to cut sensor power is a system-level parameter to be defined during detailed design.
- Deferred: OFF entry hardware shutdown sequence beyond zeroing the intensity buffers depends on the selected ADC hardware (HS2-SSM-015).
- `controlIn` is driven by `AdcsApplication`'s `sunSensorControl` output port; `AdcsApplication` sends `OFF` on entry to its own `Off` mode and `ON` on exit, per `docs/superpowers/specs/2026-05-05-adcs-hardware-manager-off-state-design.md`.

16 StarTrackerManager

Overview

`StarTrackerManager` is the Layer 2 hardware manager for the star tracker, an arcsec NV **Sagitta**. The Sagitta determines attitude on its own (camera + onboard lost-in-space and tracking algorithms) and reports it as a mounting calibrated quaternion. `StarTrackerManager`'s job is to boot the device into main firmware, load its configuration, subscribe to its solution telemetry, and cache/republish the calibrated attitude.

Requirements

ID	Requirement	Verification
HS2-STR-001	StarTrackerManager shall verify communication with the star tracker via the Ping action before considering it reachable	Inspection
HS2-STR-002	StarTrackerManager shall confirm the star tracker has booted before entering CONFIGURE, issuing a Boot action if the star tracker is still in bootloader	Inspection
HS2-STR-003	StarTrackerManager shall upload the Mounting, Camera, Distortion, Algo, and Subscription parameter sets to the star tracker's active configuration during CONFIGURE	Inspection
HS2-STR-004	StarTrackerManager shall configure the star tracker's Algo parameters for autonomous mode so the star tracker selects between LISA and tracking without ground intervention	Inspection
HS2-STR-005	StarTrackerManager shall subscribe to the Solution telemetry field so it is pushed asynchronously rather than polled	Inspection
HS2-STR-006	StarTrackerManager shall synchronize the star tracker's onboard clock via the SetTime action at CONFIGURE entry	Inspection
HS2-STR-007	StarTrackerManager shall cache the most recently received calibrated attitude quaternion, its trustworthiness flag, and its timestamp	Inspection
HS2-STR-008	StarTrackerManager shall publish the cached attitude to AdcsApplication on each new Solution arrival, and reply to AdcsApplication's requests via `getAttitude`	Inspection
HS2-STR-009	StarTrackerManager shall log a WARNING_HI event and increment a consecutive failure counter on any communication or action reply error	Inspection
HS2-STR-010	StarTrackerManager shall return to RESET after `MAX_CONSECUTIVE_COMM_ERRORS` is reached	Inspection

Design

Component Type

Active component with an internal flat F' state machine (`Fw::Sm`). All bus access goes through `LinuxUartDriver`, wired to the `ByteStreamDriverClient` port pattern.

The Omnetics connector has no dedicated hardware reset line so `StarTrackerManager` has no `resetOut/GpioWrite` port. Power sequencing of the unit itself is owned by EPS. RESET here just means waiting for the star tracker's own power on boot into its bootloader to settle.

Reference Frame & Mounting

The Sagitta's local frame has its origin at the center of the camera, X out through the aperture (boresight), Y in the mounting plane, Z completing the right hand set (Manual Figure 2.2, §2.2, p.17). The star tracker itself rotates its raw tracking solution into the spacecraft frame onboard, using a mounting quaternion that must be uploaded during CONFIGURE. The `CalibratedQuaternion_*` fields of the Solution telemetry are already expressed in the spacecraft frame. `StarTrackerManager` reads and republishes those fields directly, with no additional rotation.

Parameters

Parameter	Type	Description
`BOOT_TIMEOUT_TICKS`	`U32`	Ticks in ENABLE to reach main firmware (Version telemetry `program` = 2) before giving up and returning to RESET
`CONFIG_UPLOAD_TIMEOUT_TICKS`	`U32`	Ticks in CONFIGURE to complete the parameter upload and SetTime sequence before giving up and returning to RESET
`SETTIME_RESYNC_TICKS`	`U32`	Ticks between periodic SetTime resyncs in RUN, bounding RTC drift
`MAX_CONSECUTIVE_COMM_ERRORS`	`U32`	Consecutive comm/action reply errors in RUN before forcing a return to RESET

Ports

Port	Direction	Type	Purpose
`schedIn`	Input	`Svc.Sched`	`RateGroup3` tick (0.1 Hz). Drives the state machine's ticks and the periodic SetTime resync in RUN
`drvConnected`	Input	`Drv.ByteStreamReady`	Ready signal for the star tracker's RS485 connection
`drvReceiveIn`	Input	`Drv.ByteStreamData`	Receives raw SLIP framed bytes from `LinuxUartDriver`
`drvReceiveReturnOut`	Output	`Fw.BufferSend`	Returns ownership of the buffer arriving on `drvReceiveIn`
`drvSendOut`	Output	`Drv.ByteStreamSend`	Sends SLIP framed action and parameter requests to the star tracker
`getAttitude`	Input (sync)	`Adcs.AttitudePort`	Returns the most recently cached calibrated attitude, trust flag, and timestamp. Queried by `AdcsApplication`, which uses it to serve `DataCollectionApplication`'s attitude requests
`attitudeOut`	Output (async)	`Adcs.AttitudePort`	Pushes the cached calibrated attitude to `AdcsApplication` whenever a new Solution telemetry arrives
`cmdIn`	Input	`Fw.Cmd`	Ground commands via CmdDispatcher
`cmdResponseOut`	Output	`Fw.CmdResponse`	Command completion status
`prmGet`	Output	`Fw.PrmGet`	Load parameters from PrmDb
`logOut`	Output	`Fw.Log`	Event logging
`tlmOut`	Output	`Fw.Tlm`	Telemetry

Commands

Command Name	Parameters	Description	Response format	Postcondition
`REBOOT`	—	Issues the Reboot action (ID 7) to power cycle the star tracker's firmware. Request has no fields	Empty response	`StarTrackerManager` reenters `RESET`
`ENABLE_PROTECTION`	`wdt: bool`, `current: u8` (0 = no change, 1 = enable, 2 = disable)	Issues the EnableProtection action (ID 8) to toggle the external watchdog timer and power cycling	Empty response	Star tracker's watchdog timer and current protection set as requested
`UPLOAD_DISTORTION_CONFIG`	config reference	Reruns OpenFile (ID 33) / WriteFile (ID 35) / CloseFile (ID 34) against `ARC_FILE_CURRENT_CONFIG`.	OpenFile returns the opened file's `size: u32`. WriteFile and CloseFile responses are empty	Star tracker's active `Distortion` (ID 8) and `Camera.focallength` parameters updated to the uploaded values, produced by arcsec's in-orbit calibration procedure
`CAPTURE_CALIBRATION_FRAME`	—	Issues a Camera action (ID 15, `actionid=4`, "take a frame"). The frame must already be held in memory (`ImageProcessor.store = 1`) for Download to retrieve it. Then issues repeated Download actions (ID 9), each returning at most 1024 bytes.	Camera response is empty. Each Download response returns `position: u32` (echoed) + `data: u8[1024]`	A calibration frame is captured and downloaded, for ground troubleshooting or in-orbit diagnostics
`FORCE_RESET`	—	Forces `StarTrackerManager`'s own state machine back to RESET	N/A	`StarTrackerManager` is in `RESET`

State Machine

```
RESET
  entry: clear cached attitude/trust/timestamp, reset consecutive failure counter,
         reset wait tick counter
         load BOOT_TIMEOUT_TICKS, CONFIG_UPLOAD_TIMEOUT_TICKS, SETTIME_RESYNC_TICKS,
         MAX_CONSECUTIVE_COMM_ERRORS from PrmDb
  on tick → WAIT_RESET

WAIT_RESET
  on tick: increment wait tick counter
  if counter >= BOOT_TIMEOUT_TICKS → ENABLE

ENABLE
  entry: send Ping action (ID 0)
  on valid Ping reply (echoed ID matches) → request Version telemetry (ID 2)
    if Version.program == 1 (bootloader) → issue Boot action (ID 1, region=1)
      → rerequest Version telemetry
    if Version.program == 2 (main firmware) → CONFIGURE
```

```
  on no Ping reply, or Version.program still 1 after recheck, within BOOT_TIMEOUT_TICKS → retry
  on BOOT_TIMEOUT_TICKS exceeded without reaching main firmware → log WARNING_HI → RESET

CONFIGURE
  entry: open ARC_FILE_CURRENT_CONFIG in WRITE mode (OpenFile, ID 33)
         write Mounting (ID 6), Camera (ID 9), Distortion (ID 8),
         Algo.mode=12/AUTO with l2t/t2l thresholds (ID 16),
         Subscription.telemetry1=24/Solution (ID 18)
         via WriteFile (ID 35), then CloseFile (ID 34) to activate
  entry: issue SetTime action (ID 14) to sync the RTC
  on config upload confirmed (CloseFile reply status 0) and SetTime reply status 0 → RUN
  on CONFIG_UPLOAD_TIMEOUT_TICKS exceeded, or any action reply status != 0 → log WARNING_HI → RESET

RUN
  (continuously, off the RS485 stream): parse each incoming asynchronous_telemetry_reply
    carrying Solution telemetry (ID 24)
    cache CalibratedQuaternion_*, IsTrustworthy, StableCount, timestamp
    publish cached attitude on attitudeOut whenever a new Solution arrives
  on schedIn tick (RateGroup3, 0.1 Hz): serve getAttitude requests with the cached value;
    publish telemetry
  every SETTIME_RESYNC_TICKS: issue SetTime action (ID 14) to resync the RTC and
    bound drift
  on error (frame checksum failure, UART error, action reply status != 0):
    log WARNING_HI (throttled), increment consecutive failure count
    if consecutive failures >= MAX_CONSECUTIVE_COMM_ERRORS → RESET
```

Error recovery: `ENABLE` returns to `RESET` on boot timeout. `CONFIGURE` returns to `RESET` on upload/SetTime failure or timeout. `RUN` returns to `RESET` on consecutive comm errors.

Errors: Any action reply status other than 0, or a frame checksum failure, emits a throttled `WARNING_HI`, bumps `consecutiveFailures`, and (in RUN) triggers `RESET` once the threshold is hit.

Reference: `fprime-community/fprime-sensors ImuManager` (flat SM reference pattern), FPP flat state machines

Notes

16.1 MagnetorquerManager

MagnetorquerManager is a Layer 2 Queued worker component in the ADCS subtopology. It owns the satellite's three magnetorquer actuators (two torque rods and one air coil), converting a desired torque vector into a per-actuator direction/duty-cycle command and enforcing a lock that zeroes all output while the actuators are disabled.

Introduction

The component has no rate group and no `Reset` port; it does work only when one of its two synchronous input ports is called:

- `desiredTorqueInput` (`torqueVector_p`) — a desired torque vector from the ADCS control loop. On every call the component recomputes and re-sends all three actuators' commands.
- `setLock` (`mtToggle_p`) — requests the actuators be locked (all outputs forced to zero) or unlocked.

Actuator index `2` is treated as the air coil; indices `0` and `1` are the torque rods.

Requirements

Name	Description	Validation
MTM-001	The component shall maintain a lock state via `mtManStateMachine`, toggling between `UNLOCKED` and `LOCKED` only when a `setLock` request differs from the current state.	Unit Test
MTM-002	While `LOCKED`, the component shall drive all three actuators to zero duty cycle regardless of the last commanded torque.	Unit Test
MTM-003	On `getDesiredTorque`, the component shall attempt to produce the requested torque through the magnetorquers, maintaining torque direction even if full magnitude cannot be achieved.	Unit Test
MTM-004	The component shall log `invalidDutycyclePaRam` if any of the six parameters fails to load or update validly.	Unit Test
MTM-005	If the current measured B-field is zero, the component shall log `noMagneticField` and command all three actuators to zero duty cycle.	Unit Test

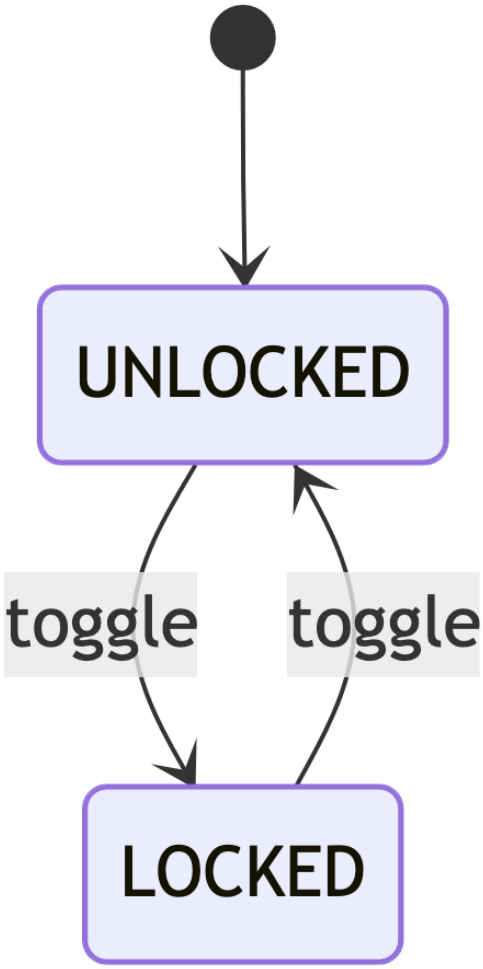
Design

Ports

Port	Kind	Direction	Type	Usage
`setLock`	sync	input	`mtToggle_p` with is_locked boolean	Requests the actuators be locked (`true`) or unlocked (`false`).
`desiredTorqueInput`	sync	input	`torqueVector_p` (`torque_vector: Vector3DBase`)	Desired torque vector; triggers a full recompute and re-send of all three actuator commands.
`getCurrentB`	—	output	`bState_p` → `Adcs.BState`	Current measured B-field (`b`, `bDot`), used to project out the non-actuatable component of torque.

Port	Kind	Direction	Type	Usage
`setDutycycle`	—	output, 3-element array	`mtDutycycle_p` with dc F32	Commanded duty-cycle magnitude for each actuator, clamped to its configured minimum and maximum.
`setDirection`	—	output, 3-element array	`mtDirection_p` with is_ high boolean	Commanded current direction for each actuator (`true` for a non-negative computed duty cycle).

State Machine



State	Meaning
`UNLOCKED`	`desiredTorqueInput` commands are written to the actuators.
`LOCKED`	All three actuators are held at zero duty cycle, regardless of the last commanded torque.

Parameters

Name	Type	Default	Applies to
TORQUE_ROD_ABS_DUTYCYCLE_MIN	F32	0.0	Actuators 0, 1
TORQUE_ROD_ABS_DUTYCYCLE_MAX	F32	1.0	Actuators 0, 1
AIR_COIL_ABS_DUTYCYCLE_MIN	F32	0.0	Actuator 2
AIR_COIL_ABS_DUTYCYCLE_MAX	F32	7.5 / 12.0 (0.625)	Actuator 2
TORQUE_ROD_MAX_MOMENT	F32	1.0 (placeholder)	Actuators 0, 1
AIR_COIL_MAX_MOMENT	F32	1.0 (placeholder)	Actuator 2

TORQUE_ROD_MAX_MOMENT / AIR_COIL_MAX_MOMENT are the dipole moment ($A \cdot m^2$) each actuator type produces at 100% duty cycle;

17 Communications Hardware Managers

`TmtcRadioManager` is the Level 2 adapter for the EnduroSat S-band radio. It translates between the communications topology and the Linux UART driver while preserving the F' communications adapter interface.

17.1 TmtcRadioManager

1. Overview

`TmtcRadioManager` is the layer 2 passive component for the Comms subtopology. It directly interacts with the Endurosat S-band transceiver through a UART interface. `The`TmtcRadioManager`is one of many layer 2 hardware managers as described in`sdd.md‘.

2. Requirements

ID	Requirement	Verification
HS2-ESM-001	TmtcRadioManager shall forward CCSDS Space Protocol telemetry packets over UART to the S-Band transceiver for downlink.	Inspection
HS2-ESM-002	TmtcRadioManager shall forward CCSDS Space Protocol command packets received over UART to the `CommandDispatcher` for processing.	Inspection
HS2-ESM-003	TmtcRadioManager shall maintain a status indicating the state of connection with the S-Band transceiver at all times	Inspection/Unit tests

3. Design

3.1 Component Type

Passive component that implements the F' `Communication Adapter Interface` that specifies both the ports and protocols used to operate with the standard F´ uplink and downlink components.

3.2 Communication Adapter Interface

Any communication component (e.g. a radio component) that is intended for use with the standard F´ uplink and downlink stack will implement the Communication Adapter Interface. This interface specifies both the ports and protocols used to operate with the standard F´ uplink and downlink components.

The communication adapter interface protocol is designed to work alongside the framer status protocol and the com queue protocol to ensure that data messages do not overload a communication interface. These protocols are discussed below.

3.2.1 Communication Queue Protocol

`Svc::ComQueue` queues messages until the communication adapter is ready to receive these messages. For each Fw::Success::SUCCESS message received, Svc::ComQueue will emit one message. Svc::ComQueue will not emit messages at any other time. This implies several things:

1. An initial Fw::Success::SUCCESS message must be sent to Svc::ComQueue to initiate data flow.
2. A Fw::Success::SUCCESS must be eventually received in response to each message for data to continue to flow.

3.2.2 Communication Framer Protocol

Framing typically happens between `Svc::ComQueue` and a communications adapter. The action taken by this protocol is dependent on the number of framed messages (frames) sent to the communication adapter in response to each message received from `Svc::ComQueue`.

The `Svc::FprimeFramer` implements the Framer status protocol by emitting a frame for each message received from `Svc::ComQueue`. Therefore, the Svc::FprimeFramer receives a status from the communication adapter on each sent frame and forwards it back to `Svc::ComQueue`. This allows the data flow to continue, or pause if the communication adapter is unable to receive more data.

Framer implementations may choose to implement a different behavior. For example, a framer may choose to accept multiple messages from Svc::ComQueue and concatenate them in a single frame. In this case, the framer implementation must manage the flow of statuses accordingly. This is summarized in the table below.

Produced Frames	Action	Rationale
0	Send one `Fw::Success::SUCCESS` status	Data must continue to flow to produce a frame
1	Pass through received status from ComQueue	Frame produced and communication adapter produces status

For HS-2, we will be utilizing the `Svc::Ccsds::SpacePacketFramer` and `Svc::Ccsds::SpacePacketDeframe`.

3.3 Ports

The communication adapter interface is composed of five ports. These ports are used to transmit outgoing data through some communication hardware and receive incoming data from that same hardware. These ports share types with the ByteStreamDriver model for backwards compatibility:

Port	Direction	Type	Purpose
`dataIn`	Input	`Svc.ComDataWithContext`	Data to be sent on wire (coming in to component)
`dataOut`	Output	`Svc.ComDataWithContext`	Data received from the wire (going out of component)
`comStatusOut`	Output	`Fw.SuccessCondition`	Status of last transmission (required for data flow)
`dataReturnOut`	Output	`Svc.ComDataWithContext`	Port returning ownership of data that came in on dataIn
`dataReturnIn`	Input	`Svc.ComDataWithContext`	Port receiving back ownership of buffer sent out on dataOut

Additionally, the `TmtcRadioManager` uses the `ByteStreamDriverClient` port interface to communicate over UART with the S-Band transciever as described in the table below:

Port	Direction	Type	Purpose
`drvConnected`	Input	`Drv.ByteStreamReady`	Ready signal for S-Band Transceiver's UART connection
`drvReceiveIn`	Input	`Drv.ByteStreamData`	Receives data from UART driver to S-Band Transceiver
`drvReceiveReturnOut`	Output	`Fw.BufferSend`	Returns ownership of buffer arriving on `drvReceiveIn`
`drvSendOut`	Output	`Drv.ByteStreamSend`	Sends data to S-Band transceiver

4. Notes

- `TmtcRadioManager` is instantiated at the top-level topology (shared with `ComCcsds` subtopology); `CommsApplication` connects to it via the top-level topology wiring.
- Detailed high-gain link configuration and `TmtcRadioManager` interface to be defined during detailed design.

18 Payload Hardware Managers

For this document, GNSS is treated as a payload/data-collection hardware manager. `GnssManager` owns receiver initialization, UART interaction, fix collection, and the position/time data consumed by higher-level components.

19 CameraManager

`CameraManager` is a Layer 2 queued component driving one physical camera. It is meant to be instantiated once per camera.

The camera hardware itself is a XIMEA xiC USB3 camera on a Sony Pregius CMOS sensor. The component mainly relies on xiAPI (https://www.ximea.com/support/wiki/apis/XiAPI_Manual) to perform all camera operations, except for image captures which happens through a GPIO pin.

Requirements

ID	Requirement	Verification
HS2-CAM-001	CameraManager shall trigger a capture after `captureImageIn` is called	Test
HS2-CAM-002	CameraManager shall report the outcome of every `captureImageIn` request via `CaptureCompleted`/`CaptureFailed` and `LastCaptureStatus`/`CapturesCompleted`	Test
HS2-CAM-003	CameraManager shall respond to `healthCheckIn` synchronously, without waiting on any in-flight `captureImageIn` delay	Test
HS2-CAM-004	CameraManager shall trigger the camera via a hardware GPIO pulse (`cameraTriggerOut`) and report `GpioTriggerError`/`CaptureFailed` if that pulse's GPIO writes do not both return `Drv.GpioStatus.OP_OK`	Test

Ports

Port	Direction	Type	Purpose
`captureImageIn`	Input, async	DataCollection.CameraCapture(imageType, delay, outputPath); types are ImageType, F32, and string	Requests one capture of `imageType`, writing the retrieved frame to `outputPath`.
`healthCheckIn`	Input, sync	`DataCollection.CameraCheckup()` returns `Types.CameraStatus`	Per-camera health check
`cameraPower`	Input, async	`DataCollection.CameraPower()` returns `Types.CameraStatus`	Camera power command
`cameraTriggerOut`	Output	Drv.GpioWrite(state); state type is Fw.Logic; returns Drv.GpioStatus	Drives the GPIO pin wired to this camera's hardware trigger input

Parameters

Name	Type	Default	Description
`CAMERA_TRIGGER_PULSE_WIDTH_S`	`F32`	`0.001`	How long `cameraTriggerOut` is held high before being pulled low

Name	Type	Default	Description
`CAMERA_DEVICE_SN`	`string`	*(empty)*	Serial number of the physical XIMEA camera this instance opens, via `xiOpenDeviceBy(XI_OPEN_BY_SN, ...)`
`CAMERA_TIMEOUT_MS`	`U32`	`2000`	how long to wait for the triggered frame before reporting `CaptureFailed`.
`CAMERA_IMAGE_DATA_FORMAT`	`U32`	`0`	xiAPI `XI_IMG_FORMAT` value for `XI_PRM_IMAGE_DATA_FORMAT` (0 = `XI_MONO8`, 5 = `XI_RAW8`, 1 = `XI_MONO16`, 6 = `XI_RAW16`)

19.1 GnssManager

1. Overview

`GnssManager` is the Layer 2 hardware manager for the GNSS receiver. It is a SkyFox Labs piNAV-NG with DROP (Dead Reckoning Orbital Propagator). Like `StarTrackerManager` and `TmtcRadioManager`, it isn't scoped to a single subtopology and instead it is at the **top-level topology** and shared across `DataCollectionApplication`, `AdcsApplication`, and `SatStateMachine`. It also feeds a PPS (Pulse Per Second) timing signal to Time services, sourced from the receiver's VPP (Valid Position Pulse) output.

`GnssManager` is an **Active** component. The bus is **UART**, 9600 baud, 8N1, LVCMOS levels. All bus access goes through `LinuxUartDriver`, wired to the `ByteStreamDriverClient` port pattern like in `TmtcRadioManager`. The receiver's RXD line is unused so there's no command channel for that.

2. Requirements

ID	Requirement	Verification
HS2-GNS-001	GnssManager shall recognize `piNAV-NG` in the receiver's postreset identification banner as confirmation the receiver is alive before entering RUN	Inspection
HS2-GNS-002	GnssManager shall continuously ingest and parse the receiver's NMEA/piNAV sentence stream as it arrives over UART	Inspection
HS2-GNS-003	GnssManager shall cache the most recent position, velocity, and GPS time	Inspection
HS2-GNS-004	GnssManager shall classify each cached fix as Autonomous, DROP Estimated, or Invalid	Inspection
HS2-GNS-005	GnssManager shall publish the cached fix and its classification to DataCollectionApplication, AdcsApplication, and SatStateMachine on each RateGroup2 tick	Inspection
HS2-GNS-006	GnssManager shall pass the receiver's VPP rising edge through as a PPS timing reference to Time services, tagged with the GPS time reported in the following LSP/LSV sentences.	Inspection
HS2-GNS-007	GnssManager shall report receiver health telemetry each tick	Inspection
HS2-GNS-008	GnssManager shall track and report the elapsed time since the last Autonomous fix, used to detect prolonged reliance on DROP Estimated or Invalid fixes	Inspection
HS2-GNS-009	GnssManager shall log a WARNING_HI event and increment a consecutive failure counter on error	Inspection
HS2-GNS-010	GnssManager shall force the receiver back into Cold Start when the elapsed time since the last Autonomous fix exceeds `DROP_MAX_AGE_TICKS`	Inspection

3. Design

3.1 Component Type

Active component and an internal flat F' state machine (`Fw::Sm`). This is a change from the other Queued pattern as the piNAV-NG streams NMEA sentences continuously at a fixed 1 Hz over UART, and the `GnssManager` needs to keep draining that stream on its own schedule rather than only when polled. `schedIn` is driven by `RateGroup2` (1 Hz) so the manager parses every incoming 1 Hz sentence group as it arrives, caches the most recent value, and reports the cached value out on each 1 s tick, matching the receiver's own update rate.

3.2 Parameters

Almost nothing about the receiver itself is changing at run. The baud rate, update rate (fixed 1 Hz), and the output sentence set are all fixed.

Parameter	Type	Description
`RESET_WAIT_TICKS`	`U32`	Ticks to wait after powering on before expecting the receiver's boot
`BANNER_WAIT_TICKS`	`U32`	Ticks to wait in ENABLE for the boot before giving up and returning to RESET
`DROP_MAX_AGE_TICKS`	`U32`	Ticks since the last fix before GnssManager forces a RESET-driven Cold Start
`MAX_CONSECUTIVE_UART_ERRORS`	`U32`	Consecutive errors in RUN before forcing a return to RESET

3.3 Ports

Port	Direction	Type	Purpose
`schedIn`	Input	`Svc.Sched`	RateGroup2 (1 Hz) tick. Drives the state machine's ticks and, in RUN, the cached fix publish
`drvConnected`	Input	`Drv.ByteStreamReady`	Ready signal for the piNAV-NG's UART connection
`drvReceiveIn`	Input	`Drv.ByteStreamData`	Receives raw NMEA/piNAV bytes from `LinuxUartDriver`
`drvReceiveReturnOut`	Output	`Fw.BufferSend`	Returns ownership of the buffer arriving on `drvReceiveIn`
`resetOut`	Output	`Drv.GpioWrite`	Drives the receiver's /RESET pin (active low), used at startup and optionally to force an early Cold Start if necessary
`vppIn`	Input (async)	`Svc.Cycle`	Fires on the VPP pin's rising edge, captured via GPIO interrupt
`getGnssFix`	Input (sync)	`Payload.gnssFixGetter_p`	Returns the most recently cached GNSS fix
`ppsOut`	Output	`Payload.gnssPps_p`	PPS record derived from the VPP pin, GPS time tagged, to Time services
`positionOut`	Output	`Payload.gnssFix_p`	Cached ECEF position, velocity, GPS time, fix classification. To DataCollectionApplication, AdcsApplication, SatStateMachine
`prmGet`	Output	`Fw.PrmGet`	Load parameters from PrmDb during RESET, and on parameter updates thereafter
`logOut`	Output	`Fw.Log`	Event logging (state transitions, fix classification changes, UART errors)
`tlmOut`	Output	`Fw.Tlm`	Telemetry (SM (State Machine) state, failure count, time since last fix, receiver voltages and current and temperature)

3.4 Commands

`GnssManager` accepts no ground commands as the receiver has no command channel. Not health monitored, same as the rest of Layer 2. All recovery is autonomous, self-healing, or escalated via telemetry.

4. State Machine

`GnssManager` uses a variant of the standard Layer 2 SM without a CONFIGURE state. The piNAV-NG has nothing to configure over the bus so this state is unnecessary. F' parameters are loaded from PrmDb during RESET instead

of a later CONFIGURE state: `WAIT_RESET` and `ENABLE` both reference parameter values (`RESET_WAIT_TICKS`, `BANNER_WAIT_TICKS`) before they run, so those values need to already be loaded by the time RESET completes.

```
RESET
  entry: flip /RESET low, hold briefly, flip hi
         clear cached fix, reset failure counter
         reset wait tick counter
         load RESET_WAIT_TICKS, BANNER_WAIT_TICKS, DROP_MAX_AGE_TICKS,
         MAX_CONSECUTIVE_UART_ERRORS from PrmDb
  on tick → WAIT_RESET

WAIT_RESET
  on tick: increment wait tick counter
           if counter >= RESET_WAIT_TICKS → ENABLE

ENABLE
  entry: begin listening on UART
  on receiving of "piNAV-NG" anywhere in a line → RUN
  on not receiving boot banner within BANNER_WAIT_TICKS → log WARNING_HI → RESET

RUN
  (continuously, based off of the UART stream): parse each incoming sentence group
    update cached position/velocity/GPS time
    classify fix as Autonomous / DROP Estimated / Invalid
    feed PPS from vppIn (VPP edge), GPS time tagged, to Time services (event driven,
    not gated by schedIn)
    if Autonomous: reset "time since last Autonomous fix" counter
    if DROP Estimated or Invalid: increment that counter
  on schedIn tick (RateGroup2, 1 Hz): publish cached fix and classification and telemetry
  on error: log WARNING_HI (throttled), increment consecutive failure count
    if consecutive failures >= MAX_CONSECUTIVE_UART_ERRORS → RESET
  on "time since last Autonomous fix" >= DROP_MAX_AGE_TICKS, but only once at least
  one Autonomous fix has ever been achieved:
    log WARNING_HI, assert /RESET to force an early Cold Start → RESET
```

Error recovery: `ENABLE` returns to `RESET` on boot banner timeout; `RUN` returns to `RESET` on consecutive UART errors or a forced Cold Start.

Errors: UART error in `ENABLE` or `RUN` emits a throttled `WARNING_HI`, bumps `consecutiveFailures`, and enters `RESET`.

Parameter reconfiguration: When `parameterUpdated()` is called while `GnssManager` is in `RUN`, it reloads all parameters from `PrmDb` in place as there is no CONFIGURE state. While in `RESET`, `WAIT_RESET`, or `ENABLE`, the signal is ignored, since parameters are freshly loaded on the next RESET entry regardless.

Reference: `fprime-community/fprime-sensors ImuManager` (flat SM reference pattern), FPP flat state machines

5. Notes

- `GnssManager` is created at the top level topology and wires out to three applications: `DataCollectionApplication` (position + time), `AdcsApplication` (position/timing, used in `AntennaPointing`), `SatStateMachine` (orbital state, sun/eclipse).
- Reset timing TBD: neither datasheet does not specify a minimum /RESET pulse width or reset settle time. It just says "no reset pulse needed," which doesn't help with the `RESET_WAIT_TICKS` number.
- Deferred: exact `consecutiveFailures` threshold and escalation path

20 Electrical Power System Hardware Managers

EPS hardware managers own the low-level board interfaces used to measure current, manage MPPT configuration, deploy panels, and service the watchdog. `EPSApplication` coordinates their information and commands at Level 3.

20.1 CurrentSensorManager

1. Overview

`CurrentSensorManager` is the Layer 2 hardware manager for the INA3221 triple-channel current/voltage monitor on the PDS (Power Distribution System) board. It owns all bus communication with the INA3221 (resetting, verifying, configuring, and reading the device) and publishes per-rail measurements to `EPSApplication` and emits them as telemetry on each rate group tick.

The INA3221 monitors three power rails (12 V, 5 V, 3.3 V), reporting bus voltage and current for each. `CurrentSensorManager` has no satellite mode awareness; it runs its startup and read loop unconditionally once initialized, driven entirely by the rate group tick. All bus access goes through the Layer 1 `LinuxI2cDriver`. `CurrentSensorManager` has no direct hardware knowledge beyond register addresses.

2. Requirements

ID	Requirement	Verification
HS2-CSM-001	CurrentSensorManager shall be the sole flight-software owner of the INA3221 over I2C.	Inspection
HS2-CSM-002	CurrentSensorManager shall read the bus voltage and current of all three rails each rate group tick.	Inspection
HS2-CSM-003	CurrentSensorManager shall publish the assembled rail measurements on railStateOut to EPSApplication each rate group tick.	Inspection
HS2-CSM-004	CurrentSensorManager shall emit telemetry channels for each rail's bus voltage and current.	Inspection
HS2-CSM-005	CurrentSensorManager shall log a WARNING_HI event and transition to RESET on any I2C bus error.	Inspection

3. Design

3.1 Component Type

Queued component with internal flat F' state machine (`Fw::Sm`).

3.2 Ports

Port	Direction	Type	Purpose
`schedIn`	Input	`Svc.Sched`	Rate group tick; drives the SM step
`busWrite`	Output	`Drv.I2c`	Write INA3221 registers (reset, configure)
`busWriteRead`	Output	`Drv.I2cWriteRead`	Write register pointer then read in one repeated-start transaction; channel reads
`railStateGet`	Input	Custom `sync_input` port returning railState struct	Per-rail bus voltage and current
`cmdIn`	Input	`Fw.Cmd`	Register-access commands via CmdDispatcher
`cmdResponseOut`	Output	`Fw.CmdResponse`	Command completion status
`logOut`	Output	`Fw.Log`	Event logging (state transitions, I2C errors)
`tlmOut`	Output	`Fw.Tlm`	Telemetry (per-rail voltage and current)

3.3 Commands

Mnemonic	Args	Description
CURRENT_SENSOR_WRITE_REG16	regAddr (INA3221Reg), value (U16)	Write to a register
CURRENT_SENSOR_SET_REG16	regAddr (INA3221Reg), mask (U16)	Read-modify-write: register OR mask
CURRENT_SENSOR_CLEAR_REG16	regAddr (INA3221Reg), mask (U16)	Read-modify-write: register AND NOT mask

CurrentSensorManager is not health-monitored. Bus-error recovery is handled autonomously by the self-healing SM.

3.4 Telemetry

Channel	Type	Description
CH1_VOLTAGE	U32	12 V rail bus voltage (mV)
CH1_CURRENT	F32	12 V rail current (mA)
CH2_VOLTAGE	U32	5 V rail bus voltage (mV)
CH2_CURRENT	F32	5 V rail current (mA)
CH3_VOLTAGE	U32	3.3 V rail bus voltage (mV)
CH3_CURRENT	F32	3.3 V rail current (mA)

4. State Machine

CurrentSensorManager uses a flat four-state F' state machine following the hardware-manager pattern: RESET → WAIT_RESET → CONFIGURE → RUN. The INA3221 has no enable step, so the ENABLE state is omitted.

```
RESET
  entry: write the RST bit in the INA3221 CONFIGURATION register to restore register defaults
  on tick → WAIT_RESET

WAIT_RESET
  on tick: one-tick settling delay for the reset to complete → CONFIGURE
           if error → RESET

CONFIGURE
  on tick: write the CONFIGURATION register to set averaging mode and conversion times
    if write OK → RUN
    if write error → log WARNING_HI → RESET

RUN
  on tick: read the bus-voltage and shunt/current registers for all three channels
           assemble railState struct
           call railStateOut port
           emit telemetry channels
           if any busWriteRead error → log WARNING_HI (throttled) → RESET
```

Error self-healing: any bus error from CONFIGURE or RUN emits a WARNING_HI event and re-enters RESET, retrying the full startup sequence on subsequent ticks.

Error recovery: any of WAIT_RESET, CONFIGURE, or RUN return directly to RESET on a bus error.

Reference: INA3221Manager (FeatherCdh), FPP flat state machines

5. Notes

- Current is derived from the INA3221 shunt-voltage registers and the per-channel shunt resistance; exact scaling and shunt values are hardware-team inputs resolved during detailed design.
- `CurrentSensorManager` is instantiated inside the EPS subtopology. Its `busWrite`/`busWriteRead` ports connect to a `LinuxI2cDriver` instance at the top-level topology; `railStateOut` connects to `EPSApplication`.
- `CurrentSensorManager` is **excluded from health monitoring** (`Svc::Health`). Only `EPSApplication` is health-checked.
- The INA3221 is PDS-dependent hardware; it is present only in the flight/PDS configuration.
- In the current hardware, Pin A0 is tied to SCL, meaning that the device I2C address is 0b1000011

20.2 DeployPanelsManager

1. Overview

`DeployPanelsManager` is a Layer 2 Active component that executes the solar panel deployment sequence when commanded by `EPSApplication`. It maintains a two-state machine tracking whether deployment has previously occurred. In both states the burn wire sequence executes; in the `DEPLOYED` state an additional `WARNING_HI` event is emitted to notify the operator that this is a re-attempt. There is no automatic retry — each deployment attempt requires an explicit `DEPLOY_PANELS` command from ground via `EPSApplication`.

2. Requirements

ID	Requirement	Verification
HS2-DPM-001	`DeployPanelsManager` shall activate the burn wire sequence upon receipt of a deploy command originating from ground.	Test
HS2-DPM-002	`DeployPanelsManager` shall emit events upon beginning and ending the burn sequence.	Test

3. Design

3.1 Component Type

Active component with internal flat F' state machine (`Fw::Sm`).

3.2 Ports

Port	Direction	Type	Purpose
`deploy`	Input	`Fw.Signal`	Deploy command forwarded from EPSApplication
`burnWire`	Output	`Drv.GpioWrite`	Burn wire GPIO; asserted for deployment duration then deasserted
`logOut`	Output	`Fw.Log`	Event logging

3.3 Commands

None. `DeployPanelsManager` is driven entirely by the `deploy` port call from `EPSApplication`, which itself accepts the ground `DEPLOY_PANELS` command.

4. State Machine

`DeployPanelsManager` uses a flat two-state F' state machine:

```
NOT_DEPLOYED
    (on deploy signal) emit deployment-initiated event
                       assert burnWire GPIO high
                       wait burn wire duration constant
                       deassert burnWire GPIO
                       emit deployment-complete event
                       transition → DEPLOYED

DEPLOYED
    (on deploy signal) emit WARNING_HI "re-attempt: panels previously deployed" event
```

```
assert burnWire GPIO high
wait burn wire duration constant
deassert burnWire GPIO
emit deployment-complete event
(remain in DEPLOYED)
```

5. Notes

- Burn wire active duration is a hardcoded constant in the component. Value TBD pending hardware team specification. This MAY eventually become a parameter stored in `PrmDb`, but we see no compelling reason for this right now.
- Burn wire duration is timed via `Os::Task::delay()` called from the deploy handler on the Active component's thread. The handler asserts the burn wire GPIO, calls `Os::Task::delay(BURN_DURATION)`, then deasserts the GPIO. No `schedIn` or timer port is required — `Os::Task::delay()` is a direct call into F' 's OS abstraction layer and blocks only the component's own thread, leaving all other components unaffected. Tradeoff: the burn cannot be cancelled mid-sequence.
- NOTE: If it turns out that our deployment has performance issues caused by having too many active threads, we can make `DeployPanelsManager` into a Queued/Passive component with a rate group for timing the burn duration.
- The `DEPLOYED` state re-executes the burn sequence to support the case where the panel did not fully deploy on the first attempt. The `WARNING_HI` event alerts the operator that this is a re-attempt.

20.3 MpptManager

1. Overview

`MpptManager` is the Layer 2 Queued hardware manager responsible for all communication with the BQ25756 MPPT and battery charging IC. No other component reads or writes the BQ25756 directly.

On each rate group tick in `RUN` state, `MpptManager` reads all relevant measurement, status, and flag registers over I2C (voltages, currents, charging status, charger/fault flags), assembles the data into a state struct, calls `EPSApplication`'s `batteryStateIn` port, and emits telemetry for the values it just read. If any abnormal charger/fault flag bit is set, it emits a `WARNING_HI` event.

`MpptManager` receives register-access commands directly from ground via the command dispatcher, split into six commands by register width and operation. Each command performs the corresponding I2C transaction against the named register and emits a confirmation event.

2. Requirements

ID	Requirement	Verification
HS2-MPT-001	MpptManager shall be the sole flight-software owner of the BQ25756 IC.	Inspection
HS2-MPT-002	MpptManager shall publish the BQ25756 measurement, charging status, and flag state on batteryStateOut each rate group tick.	Inspection
HS2-MPT-003	MpptManager shall perform the commanded register write, set, or clear transaction over I2C on receipt of a register-access command.	Inspection
HS2-MPT-004	MpptManager shall read the charger and fault flag registers each rate group tick and emit a WARNING_HI event when any abnormal status bit is set.	Inspection
HS2-MPT-005	MpptManager shall emit telemetry channels for vbatt, ibatt, vac, iac, and charging status.	Inspection

3. Design

3.1 Component Type

Queued component with internal flat F' state machine (`Fw::Sm`).

3.2 Ports

Port	Direction	Type	Purpose
`schedIn`	Input	`Svc.Sched`	Rate group tick; drives the SM step (measurement read cycle in RUN)
`busWrite`	Output	`Drv.I2c`	Write BQ25756 registers (reset, configure, register-access commands)
`busWriteRead`	Output	`Drv.I2cWriteRead`	Write register pointer then read in one repeated-start transaction; all measurement, status, and flag reads
`batteryStateGet`	Input	Custom `sync_input` port returning batteryState struct	Battery and IC state (vbatt, ibatt, vac, iac, vfb, temperature, charging status, charger/fault flags, MPPT state)
`cmdIn`	Input	`Fw.Cmd`	Register-access commands via CmdDispatcher
`cmdResponseOut`	Output	`Fw.CmdResponse`	Command completion status
`logOut`	Output	`Fw.Log`	Event logging
`tlmOut`	Output	`Fw.Tlm`	Telemetry (vbatt, ibatt, vac, iac, charging status)

Port	Direction	Type	Purpose

3.3 Commands

Each command names its target register through a width-specific enum (`BQ25756Reg8` or `BQ25756Reg16`), so ground can only address a register of the matching width.

Mnemonic	Args	Description
`MPPT_WRITE_REG8`	regAddr (BQ25756Reg8), value (U8)	Write to an 8-bit register
`MPPT_SET_REG8`	regAddr (BQ25756Reg8), mask (U8)	Read-modify-write: register OR mask
`MPPT_CLEAR_REG8`	regAddr (BQ25756Reg8), mask (U8)	Read-modify-write: register AND NOT mask
`MPPT_WRITE_REG16`	regAddr (BQ25756Reg16), value (U16)	Write to a 16-bit register
`MPPT_SET_REG16`	regAddr (BQ25756Reg16), mask (U16)	Read-modify-write: register OR mask
`MPPT_CLEAR_REG16`	regAddr (BQ25756Reg16), mask (U16)	Read-modify-write: register AND NOT mask

On success each command emits an activity event carrying the operation, register, and the byte/halfword actually written (the post-modify value for set/clear). On I2C failure it emits a `WARNING_HI` write-error event and returns an `EXECUTION_ERROR` command response.

3.4 Telemetry

Channel	Type	Description
`VBATT_MV`	`U32`	Battery voltage in mV (VBAT_ADC × 2)
`IBATT_RAW`	`U16`	Battery current raw ADC (scaling TBD)
`VAC_MV`	`U32`	Input voltage in mV (VAC_ADC × 2)
`IAC_RAW`	`U16`	Input current raw ADC (scaling TBD)
`CHARGING_STATE`	`BQ25756EChargingState`	Charging state from CHARGER_STATUS_1 bits 2:0

4. State Machine

`MpptManager` uses a flat four-state F' state machine: `RESET → WAIT_RESET → CONFIGURE → RUN`. The BQ25756 has no enable step (it does Max Power Point Tracking by default upon being turned on), so the standard `ENABLE` state is omitted.

```
RESET
  on tick: write REG_RST bit in POW_PATH_REV_CONT (0x19 bit 7) to restore register defaults
    if busWrite OK → WAIT_RESET
    if busWrite error → log WARNING_HI, remain in RESET (retry next tick)

WAIT_RESET
  on tick: settling delay for the register reset to complete → CONFIGURE

CONFIGURE
  entry/on tick: write ADC_CONT (0x2B) to enable the ADC in continuous mode
    if all writes OK → RUN
    if any write error → log WARNING_HI → RESET
```

```
RUN
  on tick: read measurement registers (VBAT_ADC, IBAT_ADC, VAC_ADC, IAC_ADC, VFB_ADC, TS_ADC)
           read charging status register (CHARGER_STATUS_1)
           read charger/fault flag registers (CHARGER_FLAG_1, CHARGER_FLAG_2, FAULT_STATUS,
            ↪ FAULT_FLAG)
           assemble batteryState struct
           call batteryStateOut port
           emit telemetry channels
           if any abnormal charger/fault flag bit is set → emit WARNING_HI flag event (throttled)
           if any busWriteRead error → log WARNING_HI (throttled) → RESET
  on register-access command (MPPT_WRITE_REG8 / MPPT_SET_REG8 / MPPT_CLEAR_REG8 /
                              MPPT_WRITE_REG16 / MPPT_SET_REG16 / MPPT_CLEAR_REG16):
           perform the I2C write (read-modify-write for set/clear)
           emit register-written event, or WARNING_HI write-error event on I2C failure
```

Fault handling is done by the per-tick flag read: `CHARGER_FLAG_1/2` and `FAULT_FLAG` are cleared-on-read, so each tick captures the events that occurred since the previous tick. An abnormal flag emits a warning event but does not change state; the IC continues running. Only I2C bus errors self-heal by returning to `RESET`.

Command behavior outside RUN: register-access commands received while in `RESET`, `WAIT_RESET`, or `CONFIGURE` are queued and execute once the component processes them. Ordering against the reset/configure writes is TBD during detailed design.

Error recovery: `RESET` retries itself on a busWrite error; `CONFIGURE` and `RUN` return to `RESET` on their own bus errors.

Reference: FPP flat state machines

5. Notes

- `batteryStateOut` port type is a custom struct carrying all BQ25756 measurement and status data plus the charger/fault flags. Consider splitting into a measurements port and a flags port if the struct becomes unwieldy; see `EPSApplication` notes for the matching `batteryStateIn` discussion.
- `MpptManager` owns and publishes the periodic measurement telemetry (`VBATT_MV`, `IBATT_RAW`, `VAC_MV`, `IAC_RAW`, `CHARGING_STATE`). It still forwards the same values to `EPSApplication` on `batteryStateOut`; `EPSApplication` consumes them for `powerState` and may use them for more complex logic in the future.
- The ADC must be enabled before measurement reads return real values; enabling it in `CONFIGURE` guarantees `RUN` reads are valid without an operator step.
- The set of "abnormal" _FLAG bits that warrant a warning event is TBD pending a detailed pass over the BQ25756 datasheet fault/flag register definitions.
- `BQ25756Reg8` / `BQ25756Reg16` enums enumerate all addressable registers by name and width; the `BQ25756RegOp` enum names the write/set/clear operation carried in the confirmation events.

20.4 WatchdogPinger

1. Overview

`WatchdogPinger` is a Layer 2 Passive component that satisfies the hardware watchdog on the EPS board by toggling a dedicated GPIO pin on each rate group tick. It has no satellite mode awareness and no state machine.

2. Requirements

ID	Requirement	Verification
HS2-WDP-001	WatchdogPinger shall pulse the watchdog GPIO on each rate group tick.	Test

3. Design

3.1 Component Type

Passive component. No queue, no thread, no state machine.

3.2 Ports

Port	Direction	Type	Purpose
`schedIn`	Input	`Svc.Sched`	Rate group tick; triggers watchdog GPIO toggle
`watchdogPing`	Output	`Drv.GpioWrite`	GPIO pin connected to EPS board hardware watchdog

3.3 Commands

None.

4. Operational Behavior

On each `schedIn` tick, `WatchdogPinger` asserts then deasserts `watchdogPing` to produce a pulse satisfying the EPS hardware watchdog timer. If this component stops being scheduled — due to a software hang or rate group slip — the hardware watchdog will expire and reset the system.

5. Notes

- The watchdog timer we plan on using (TI product number TPS3431SDRBR) requires a minimum pulse width of only 50 ns to reset the timer. Thus, it is not required to have any OS-created delay within this component. We can instead simply call the GPIO driver to pull WDI high and then low sequentially.
- Tick rate and GPIO pulse timing to be confirmed with the hardware team against the specific watchdog IC's timeout and minimum pulse width specifications.
- `WatchdogPinger` is scheduled via the standard F' rate group mechanism. The rate group must be selected such that its period is comfortably within the watchdog timeout window including any rate group slip margin.

21 Thermal Hardware Managers

The thermal hardware-manager layer reads temperature sensors and drives heaters. The managers isolate Linux I2C and PWM details from `ThermalApplication`.

21.1 HeaterManager

HeaterManager is a Layer 2 Queued worker component in the Thermal subtopology. It owns the PWM-driven heater output channel, bringing the channel up through an internal state machine and translating duty-cycle commands from `ThermalApplication` into PWM waveforms.

Introduction

The component is driven by three synchronous input ports:

- `schedIn` (`Svc.Sched`) — a rate-group tick
- `heaterCmdIn` (`Thermals.HeaterCmd`) — a duty-cycle command in percent. Commands are dropped (logging `HeaterCmdDropped`) unless the state machine is in `RUN`.
- `manualReset` (`Thermals.ManualReset`) — forces an immediate return to `RESET` from `CONFIGURE`/`RUN`, which disables the PWM channel as part of `RESET`'s own entry actions. Lets the owning application request a fresh `powerAllowed` check without waiting for a PWM fault.

The channel itself is a Linux sysfs PWM channel owned by `Drv::LinuxPwmDriver` (`FlightComputer/Drivers/LinuxPwm`), reached over three synchronous output ports: `pwmSetPeriodOut`, `pwmSetDutyCycleOut`, and `pwmEnableOut` (all `Drv.Pwm*`, returning `Drv.PwmStatus`). The driver writes `period`, `duty_cycle`, and `enable` sysfs files under `pwmchip<N>/pwm<N>`.

Before start-up can proceed past `RESET`, the component asks the owning application for permission via `powerAllowed`.

Requirements

ID	Requirement	Verification
HS2-HTM-001	HeaterManager shall set the configured PWM period during CONFIGURE	Unit test
HS2-HTM-002	HeaterManager shall enable the PWM channel at the end of CONFIGURE	Unit Test
HS2-HTM-003	HeaterManager shall serve heaterCmdIn requests by writing the commanded duty cycle to LinuxPwmDriver in RUN state	Unit test
HS2-HTM-004	HeaterManager shall clamp heater output to zero in all non-RUN states	Unit test
HS2-HTM-005	HeaterManager shall emit WARNING_HI, disable the PWM channel, and self-heal to RESET on any PWM write failure	Unit test
HS2-HTM-006	HeaterManager shall log `HeaterCmdDropped` and take no other action when `heaterCmdIn` is received outside `RUN`	Unit test
HS2-HTM-007	HeaterManager shall force an immediate transition to `RESET` when `manualReset` is invoked from `CONFIGURE` or `RUN`	Unit test
HS2-HTM-008	HeaterManager shall zero the duty cycle (both the PWM driver write and its telemetry) on every entry to RESET - the only place this happens	Unit test
HS2-HTM-009	HeaterManager shall disable the PWM channel on every entry to RESET - the only place this happens	Unit test

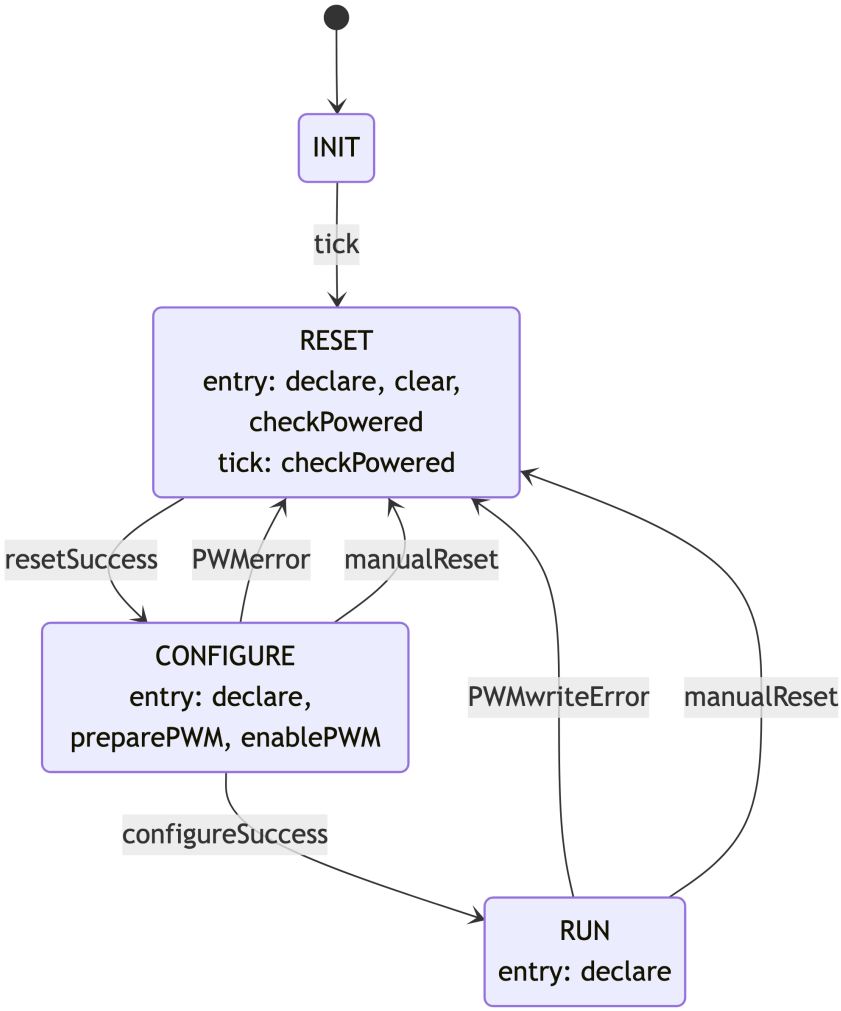
Design

Ports

Port	Kind	Direction	Type	Usage
`schedIn`	sync	input	`Svc.Sched`	Rate-group tick; sends `tick` to the state machine on every call.
`heaterCmdIn`	sync	input	`Thermals.HeaterCmd`	Duty-cycle command in percent.
`pwmSetPeriodOut`	sync	output	`Drv.PwmSetPeriod`	Sets the channel period in nanoseconds; called once, during `CONFIGURE`.
`pwmSetDutyCycleOut`	sync	output	`Drv.PwmSetDutyCycle`	Sets duty cycle in nanoseconds; called to zero the output on every entry to `RESET` and on every accepted `RUN` command.
`pwmEnableOut`	sync	output	`Drv.PwmEnable`	Enables (`HIGH`, in `CONFIGURE`) or disables (`LOW`, on every entry to `RESET`, via `clear`) the channel.
`powerAllowed`	sync	output	`Thermals.PowerAllowed`	Queries whether the application currently allows the heater to power on; polled once per `tick` while in `RESET`.
`manualReset`	sync	input	`Thermals.ManualReset`	Forces the state machine back to `RESET` immediately (which disables the PWM channel as part of `RESET`'s own entry actions, if not already there); lets the owning application request a fresh `powerAllowed` check without waiting for a PWM fault.
`timeCaller`, `Fw.Event`, `Fw.Channel`	standard AC ports	—	—	Boilerplate event/telemetry/time wiring. No `Fw.Command` import — this component defines no commands.

State Machine

`heaterManagerSM` (`Thermals_HeaterManagerStateMachine_t`, defined in `HeaterManagerStateMachine.fpp`) owns start-up and fault recovery. Its current state mirrors the `State` telemetry channel via the `HeaterManagerState` enum.



Each state's behavior, in the order things actually happen (`declare` — updating the cached `SMstate` member, the `State` telemetry channel, and logging `StateChanged` — runs on entry to every state and is omitted below to avoid repeating it four times; it also logs `SMsignalInvalid` if handed an unexpected/initial-transition signal):

```
INIT
  on tick → RESET

RESET
  entry: clear - disable the PWM channel (pwmEnableOut(0, LOW)); the only place this
           happens (there's no separate "bail" action - every path back to RESET,
           whatever the cause, disables the channel right here as part of arriving)
           on failure → log PwmDisableFailed (channel may still be enabled; no retry)
         then zero the duty cycle (pwmSetDutyCycleOut(0, 0)) and its telemetry
           (DutyCycleNs/DutyPercent); also the only place this happens
           on failure → log PwmGeneralFailure (stay in RESET either way)
  on tick: checkPowered - ask the owning application via powerAllowed
             if ON → CONFIGURE
             otherwise stay in RESET; ask again next tick

CONFIGURE
  entry: preparePWM - set the period (pwmSetPeriodOut(0, PERIOD_NS))
           on failure → log PwmGeneralFailure → RESET
         enablePWM - enable the channel (pwmEnableOut(0, HIGH))
           on failure → log PwmGeneralFailure → RESET
           on success → RUN
  on manualReset → RESET
```

```
RUN
  on heaterCmdIn(dutyPercent):
    clamp dutyPercent to [0, 100] (log DutyClamped if clamped)
    write pwmSetDutyCycleOut
      on failure → log PwmWriteFailed, re-zero locally, → RESET
  on manualReset → RESET
```

Every → RESET transition above disables the channel as a side effect of arriving there (via clear, above)

Dropped commands: heaterCmdIn received outside RUN is dropped, not queued or replayed — logs HeaterCmdDropped (activity low) and takes no other action.

schedIn_handler sends tick unconditionally on every call.

Telemetry

Name	Type	Notes
State	Thermals.HeaterManagerState	Mirrors the state machine's current state.
DutyPercent	F32	Last accepted (possibly clamped) duty cycle, in percent. Reset to 0 on every entry to RESET, then updated on each accepted RUN command.
DutyCycleNs	U32	Last accepted duty cycle, in PWM nanoseconds (dutyPercent/100 * PERIOD_NS).

Events

Name	Severity	Purpose
StateChanged	activity high	Emitted on every state-machine transition.
DutyClamped	activity high	A heaterCmdIn command was clamped to the 0-100 percent range.
PwmWriteFailed	warning high	The RUN-state duty-cycle write failed.
PwmGeneralFailure	warning high	A PWM call failed either zeroing the duty cycle on entry to RESET or during CONFIGURE start-up; carries the state it happened in. (Not emitted for a channel-disable failure — see PwmDisableFailed.)
PwmDisableFailed	warning high	The channel-disable write, attempted on every entry to RESET (via clear), itself failed — the channel may still be enabled.
SMsignalInvalid	warning high	An unrecognized or initial-transition signal reached the state machine — indicates a state-machine bug, not a PWM fault.
HeaterCmdDropped	activity low	A heaterCmdIn command arrived while not in RUN and was dropped.

21.2 TemperatureSensorManager

TemperatureSensorManager is a Layer 2 Queued worker component in the Thermal subtopology. It owns all onboard temperature sensors, sequencing their power-up and configuration through an internal state machine, polling them once per rate-group tick.

The component is driven by two synchronous input ports:

- `schedIn` (`Svc.Sched`) — a rate-group tick. All state-machine progression and sensor polling happens here.
- `tempReadIn` (`Thermals.TempRead`, sized `[NUM_TEMPERATURE_SENSORS]`) — lets other components pull the most recently cached readings without waiting on the bus.

Requirements

Name	Description	Validation
TSM-001	While in `WAIT_RESET`, the component shall wait `WAIT_TICKS` scheduler ticks before signaling `waitSuccess` and advancing to `CONFIGURE`.	Unit Test
TSM-002	While in `RUN`, the component shall poll all `NUM_TEMPERATURE_SENSORS` sensors once per `schedIn` tick and publish the results as the `temperature` telemetry channel.	Unit Test
TSM-003	The component shall serve the last cached readings on `tempReadIn` regardless of state, invalidating all entries (setting each reading's valid flag to false) whenever the state machine (re-)enters any state other than `RUN`.	Unit Test
TSM-004	On any bus error, the component shall log a warning, invalidate all cached readings (setting each reading's valid flag to false), and return to `RESET`.	Unit Test
TSM-005	While in `CONFIGURE`, the component shall write each sensor's Configuration Register 0 (Automatic Conversion mode) and Configuration Register 1 (Type K, no averaging) over SPI.	Unit Test
TSM-006	While in `RUN`, the component shall skip that sensor's SPI read on ticks where `DRDY` reports not-ready.	Unit Test
TSM-07	`setSensorSkip` with `resetComponent` set shall drive the state machine back to `RESET` (from any of `WAIT_RESET`, `CONFIGURE`, `RUN`) without logging `SMsignalInvalid`, invalidating all cached readings in the process.	Unit Test

Design

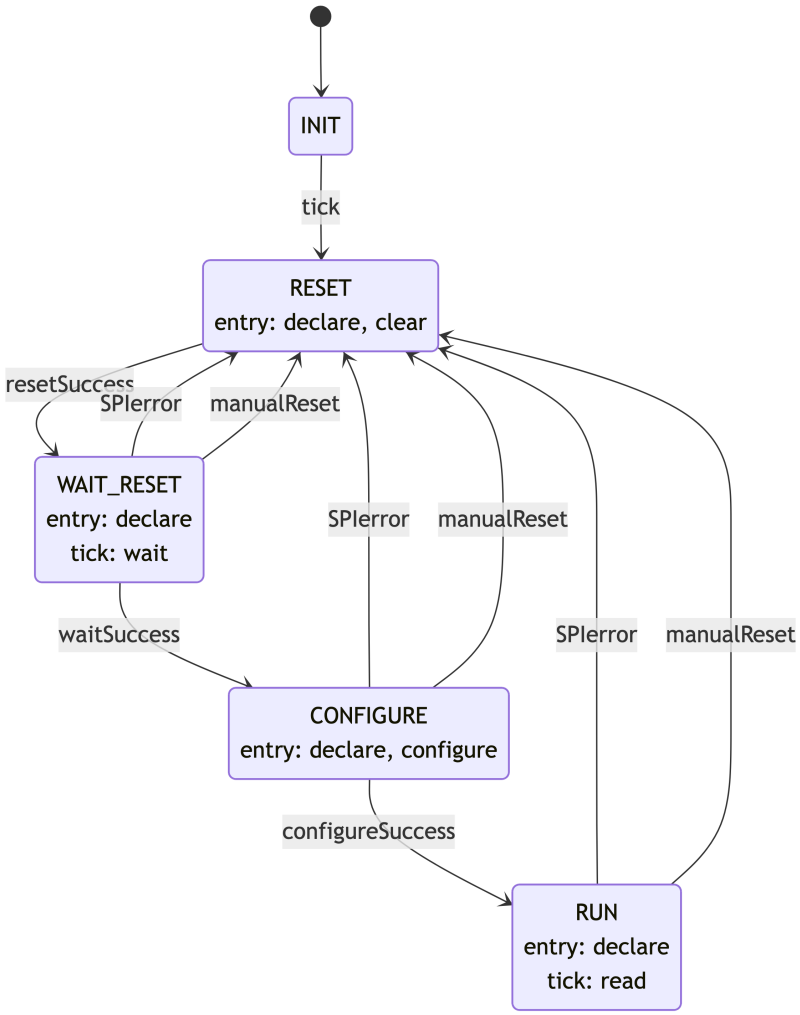
Ports

Port	Kind	Direction	Type	Usage
`schedIn`	sync	input	`Svc.Sched`	Rate-group tick; drives state-machine progression and sensor polling.
`tempReadIn`	sync	input, one per temperature sensor	`Thermals.TempRead`	Returns the last cached `TemperatureReadings` to a caller
`spiWriteRead`	—	output	`Drv.SpiWriteRead`	SPI transaction (address byte + data byte(s)) used to read/write each sensor's registers.

Port	Kind	Direction	Type	Usage
`sensorSelect`	—	output, one per sensor-select pin	`Drv.GpioWrite`	Binary-encoded GPIO select lines addressing which sensor is active on the shared SPI bus for the current transaction.
`sensorDRDY`	—	output	`Drv.GpioRead`	Signal from amplifier for whether the selected sensor is safe to read

State Machine

`tempSenseManagerSM` (`Thermals_TemperatureSensorManagerStateMachine_t`, defined in `TemperatureSensorManagerStateMachine.fpp`) owns the bring-up and fault-recovery sequence. Its current state mirrors the `State` telemetry channel via the `TemperatureSensorManagerState` enum.




```
INIT
  on tick → RESET

RESET
  entry: clear - reset the wait-tick counter to 0, log cleared
         → WAIT_RESET immediately (clear unconditionally signals resetSuccess - RESET is
           effectively a one-tick pass-through, not a place that waits for anything itself)

WAIT_RESET
```

```
  on tick: wait - increment the wait-tick counter, telemeter ticksWaited
             once counter >= WAIT_TICKS → CONFIGURE
  on SPIerror → RESET
  on manualReset → RESET

CONFIGURE
  entry: configure - for each sensor not marked skipped (via setSensorSkip):
           write Configuration Register 0 (CMODE=1, Automatic Conversion mode) and
           Configuration Register 1 (TC_TYPE=0011 Type K, AVGSEL=000 no averaging) in one
           multi-byte SPI transaction
           on SPI error (any sensor) → log SPIError → RESET immediately, without
             configuring the remaining sensors
           on success (all non-skipped sensors) → log configured → RUN
         a skipped sensor receives no SPI traffic at all
  on SPIerror → RESET
  on manualReset → RESET

RUN
  on tick: read - for each sensor:
             if marked skipped (via setSensorSkip) → ignored entirely (no select, no SPI)
             else if a connected sensorDRDY reports not-ready → skip this sensor this tick
             else → SPI read; update cached reading + per-sensor nominality; publish to
               the temperature telemetry channel; emit Overheating/Cold/Nominal on a
               threshold crossing (edge-triggered, not every tick)
           a sensor skipped for either reason (setSensorSkip or not-ready DRDY) is left out
           of temperature for that tick - its entry publishes as invalid rather than
           repeating a stale value
  on SPIerror → RESET
  on manualReset → RESET
```

Part IV

Level 1 - Drivers

22 Operating-System Abstraction Layer (OSAL)

22.1 F' OSAL and Linux on the BeagleBone Black

F' uses an operating-system abstraction layer (OSAL) so framework code can request common services without embedding a particular operating system into every component. The OSAL provides wrappers for tasks, queues, mutexes, time, files, directories, and related operating-system resources. The Level 1 driver boundary similarly provides typed F' interfaces for Linux serial and bus access.

On the BeagleBone Black, these abstractions are implemented with Linux and POSIX facilities. Flight components therefore use F' and OSAL interfaces for scheduling, synchronization, timing, and file access, while Linux-specific details remain in the driver or deployment layer. This keeps the application and hardware-manager code testable in GDS and on a host system, while still allowing the deployed topology to use the BeagleBone's UART, I2C, SPI, GPIO, PWM, and filesystem resources.

The OSAL is not a substitute for a device manager. It can open a file or serial endpoint and provide a task or queue, but a Level 2 manager still owns the device protocol, register meanings, initialization sequence, and hardware-specific recovery policy.

22.2 Os::File

`Os::File` provides the F' OSAL interface for opening, reading, writing, seeking, flushing, and closing files. On the BeagleBone Black, the implementation maps these operations to Linux filesystem calls. HuskySat-2 uses this boundary for configuration, stored data, data products, and file-based communications services; file policy and data ownership remain with the application or service that uses the file.

Reference: F' OSAL source

22.3 Os::Task

`Os::Task` creates and controls execution contexts through a portable F' OSAL interface. The Linux implementation supplies the underlying POSIX thread behavior. HuskySat-2 active components and service components use this abstraction so deployment-specific priorities, stack sizes, and startup behavior remain in the topology configuration.

Reference: F' OSAL source

22.4 Os::Queue

`Os::Queue` provides the message-passing primitive used by active and queued components. On Linux, the implementation supplies the synchronization needed to enqueue, block, and dequeue work safely. The queue separates command arrival from component execution and helps prevent long-running work from blocking synchronous callers.

Reference: F' OSAL source

22.5 Os::Mutex

`Os::Mutex` provides portable mutual exclusion for shared state. The Linux implementation maps to the platform's synchronization primitives. Components will keep protected regions short and use F' guarded ports or component-owned synchronization where that makes the ownership of shared state clearer.

Reference: F' OSAL source

22.6 Os::Time

`Os::Time` provides portable access to clocks, timestamps, and time-related operations. On the BeagleBone Black, the implementation uses Linux clocks while preserving F' time types at component interfaces. This supports telemetry timestamps, scheduling, timeout handling, and tests that substitute controlled time sources.

Reference: F' OSAL source

23 LinuxPwmDriver

1. Overview

`LinuxPwmDriver` is a Layer 1 passive driver for one Linux PWM channel. It gives a hardware manager (Layer 2 component) a synchronous F' interface for setting the channel period and duty cycle and for enabling or disabling the output. It contains no device or actuator logic.

It uses the Linux PWM sysfs interface (`/sys/class/pwm/pwmchipN/pwmM/`). One driver instance owns one channel.

2. Requirements

ID	Requirement	Verification
HS2-PWM-001	LinuxPwmDriver shall open and own one configured PWM channel during topology setup.	Inspection
HS2-PWM-002	LinuxPwmDriver shall provide synchronous operations to set the channel period, set the duty cycle, and enable or disable the channel.	Inspection
HS2-PWM-003	LinuxPwmDriver shall reject a duty cycle greater than the most recently applied period without writing the duty-cycle value.	Inspection
HS2-PWM-004	LinuxPwmDriver shall report an open failure, an invalid request, or a runtime write failure to its caller.	Inspection
HS2-PWM-005	LinuxPwmDriver shall report open and runtime failures through F' events.	Inspection

3. Design

3.1 Component Type

Passive component. Its port handlers run synchronously on the calling thread. It has no state machine, commands, parameters, telemetry channels, or health-monitoring interface. The driver retains only its open state and the last successfully applied period.

3.2 Ports

The following project-defined ports form the `Drv.Pwm` interface.

Port	Direction	Type	Purpose
`pwmSetPeriod`	Input (sync)	`Drv.PwmSetPeriod`	Set the period in nanoseconds.
`pwmSetDutyCycle`	Input (sync)	`Drv.PwmSetDutyCycle`	Set the duty cycle in nanoseconds.
`pwmEnable`	Input (sync)	`Drv.PwmEnable`	Enable or disable the channel.
`logOut`	Output	`Fw.Log`	Report open and write failures.

Each operation returns `Drv.PwmStatus`.

Status	Meaning
`PWM_OK`	Operation succeeded.
`PWM_NOT_OPENED`	The channel was not successfully opened.
`PWM_OPEN_ERR`	Exporting or opening the channel failed.
`PWM_WRITE_ERR`	A write to the PWM interface failed.
`PWM_INVALID_PARAM`	The requested duty cycle exceeds the configured period.
`PWM_OTHER_ERR`	An unexpected driver error occurred.

3.3 Configuration and Operation

Topology setup calls `open(chipNum, channelNum)` before the connected manager makes any port calls. The driver exports the selected channel when necessary and opens its `period`, `duty_cycle`, and `enable` entries. If setup fails, port calls return `PWM_NOT_OPENED`.

A manager configures a channel in this order:

1. Set the period.
2. Set a duty cycle no greater than that period.
3. Enable the channel.

The manager owns actuator-safe behavior. For example, duty cycle will be set to 0 before enabling a channel or when dealing with an error.

24 Drv::LinuxUartDriver

Overview

`Drv::LinuxUartDriver` is an F' Level 1 passive driver that exposes a Linux serial device through F' byte-stream ports. It owns operating-system access to a configured UART but does not interpret the protocol carried over that UART.

HuskySat-2 Use

HuskySat-2 uses UART access for the EnduroSat S-band radio, the GNSS receiver, and other serial peripherals. `TmtcRadioManager` and `GnssManager` own the device-specific protocols above this driver.

Component Type and Integration

The driver is passive and synchronous. A topology instance configures the Linux device path and serial settings, then connects its byte-stream ports to a hardware manager or communications component. Protocol framing, packet validation, retries, and mission state remain outside the driver.

F' Reference

LinuxUartDriver source and interface

25 Drv::LinuxI2cDriver

Overview

`Drv::LinuxI2cDriver` is an F' Level 1 passive driver that exposes Linux I2C transactions through typed F' ports. It provides bus access while leaving register maps, device initialization, and recovery policy to the connected hardware manager.

HuskySat-2 Use

The EPS board, temperature sensors, sun-sensor support hardware, and any I2C-configured IMU use this driver. Their managers validate device responses and translate register values into subsystem data.

Component Type and Integration

The driver is passive and synchronous. A topology instance selects the Linux I2C bus and target address, then connects read, write, or combined transaction ports to a Level 2 manager. It does not contain device-specific commands or telemetry.

F' Reference

LinuxI2cDriver source and interface

26 Drv::LinuxSpiDriver

Overview

`Drv::LinuxSpiDriver` is an F' Level 1 passive driver that exposes Linux SPI transactions through typed F' ports. It owns access to a configured SPI device but does not interpret the bytes exchanged with that device.

HuskySat-2 Use

The IMU and sun-sensor ADC use SPI-backed interfaces in the current hardware-manager designs. Their Level 2 managers own chip-specific transaction formats and sequencing; the current SSD and microSD storage architecture uses Linux filesystem services rather than an SPI NOR storage path.

Component Type and Integration

The driver is passive and synchronous. A topology instance configures the Linux SPI device and connects transaction ports to a manager. Chip-select, register interpretation, initialization, and retry behavior are owned by the higher-level component that uses the bus.

F' Reference

LinuxSpiDriver source and interface

27 Drv::LinuxGpioDriver

Overview

`Drv::LinuxGpioDriver` is an F' Level 1 passive driver for Linux GPIO lines. It exposes digital inputs and outputs without assigning meaning to a particular enable, reset, interrupt, or status signal.

HuskySat-2 Use

GPIO lines support hardware enables, reset lines, chip-select support, and discrete status signals throughout the ADCS, EPS, communications, and payload subsystems. The connected manager owns the safe value and the interpretation of each line.

Component Type and Integration

The driver is passive and synchronous. Each topology instance is configured for a Linux GPIO line and connects to the manager that owns the associated hardware behavior. Device sequencing and fault recovery remain above the driver layer.

F' Reference

LinuxGpioDriver source and interface

Part V

Appendix

28 Hardware and Platform Inventory

The BeagleBone Black is the on-board Linux flight computer. It runs the F' deployment, hosts the spacecraft software topology, and accesses spacecraft hardware through Linux-backed F' drivers. Linux provides the operating-system abstractions for UART, I2C, SPI, GPIO, PWM, timing, USB, and file storage; device-specific behavior belongs in the higher-level manager or application that owns the hardware.

The inventory below is derived from the current hardware-manager interfaces.

Hardware or device	Interface to BeagleBone Black	Owning software
LOST camera	USB	`DataCollectionApplication`
FOUND camera	USB	`DataCollectionApplication`
Arcsec Sagitta Star Tracker	UART; `StarTrackerManager`	ADCS and data collection
SkyFox piNAV-NG GNSS receiver	UART; reset GPIO; VPP/PPS GPIO input	`GnssManager`
VectorNav VN-100 IMU/AHRS	RS-232/UART	`ImmuManager`
Six sun photodiodes and MCP3208 ADC	SPI; chip-select GPIO	`SunSensorManager`
Two torque rods and one air coil	PWM duty cycle and direction outputs	`MagnetorquerManager`
INA3221 three-channel current monitor	I2C	`CurrentSensorManager`
BQ25756 MPPT/battery charger	I2C	`MpptManager`
Solar-panel deployment burn wire	GPIO	`DeployPanelsManager`
EPS hardware watchdog	GPIO pulse	`WatchdogPinger`
EnduroSat S-band transceiver	UART	`TmtcRadioManager`
Temperature sensors	SPI; sensor-select GPIO; DRDY input	`TemperatureSensorManager`
Heaters	PWM	`HeaterManager`
microSD card and external SSD	Linux filesystem services	F' file-management and data-product services; mission applications

The BeagleBone Black and Linux are platform services rather than mission components. The driver layer exposes their operating-system interfaces; hardware managers interpret device registers and protocols; applications coordinate subsystem behavior.

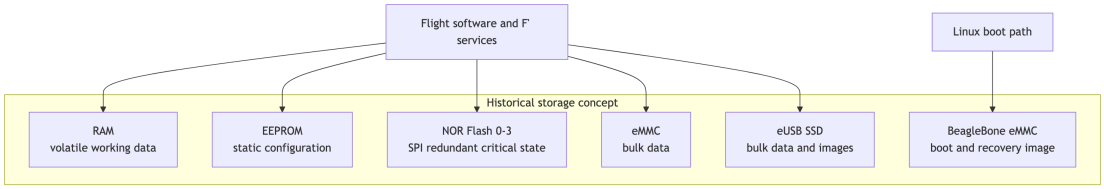
29 Radiation and Storage Architecture

This section describes how HuskySat-2 stores flight-software state, mission data, and recovery images, and how the software will detect and recover from faults that could be caused by radiation or other single-event disturbances. The radiation environment and final hardware assurance analysis are not yet baselined, so the approach below describes the current design direction and the software controls that can be verified independently of the final dose and upset-rate analysis.

29.1 Historical Architecture

The original flight-computer memory concept separated data by volatility, criticality, and recovery value. Volatile working data was placed in RAM, static configuration was assigned to EEPROM, mission-critical state was assigned to three redundant NOR flash devices connected over SPI, and bulk data was assigned to eMMC and an eUSB SSD. The BeagleBone Black's internal eMMC was also considered for the bootloader, Linux kernel, root file system, and a flight-software recovery image.

The three NOR devices were intended to provide redundancy for state and fault data: one device would be active at a time, while the software tracked device health, wear information, and the status of the stored software copy. A decision flow separated data that could be regenerated from data that needed to survive a reboot or be recovered after a fault. This architecture provided a clear place for duplicated mission state and recovery metadata, but it depended on interfaces and boot paths that were difficult to realize on the selected BeagleBone Black configuration.



The historical architecture is retained in this SDD as design rationale and as a record of the intended radiation and redundancy considerations. It is not the current flight configuration.

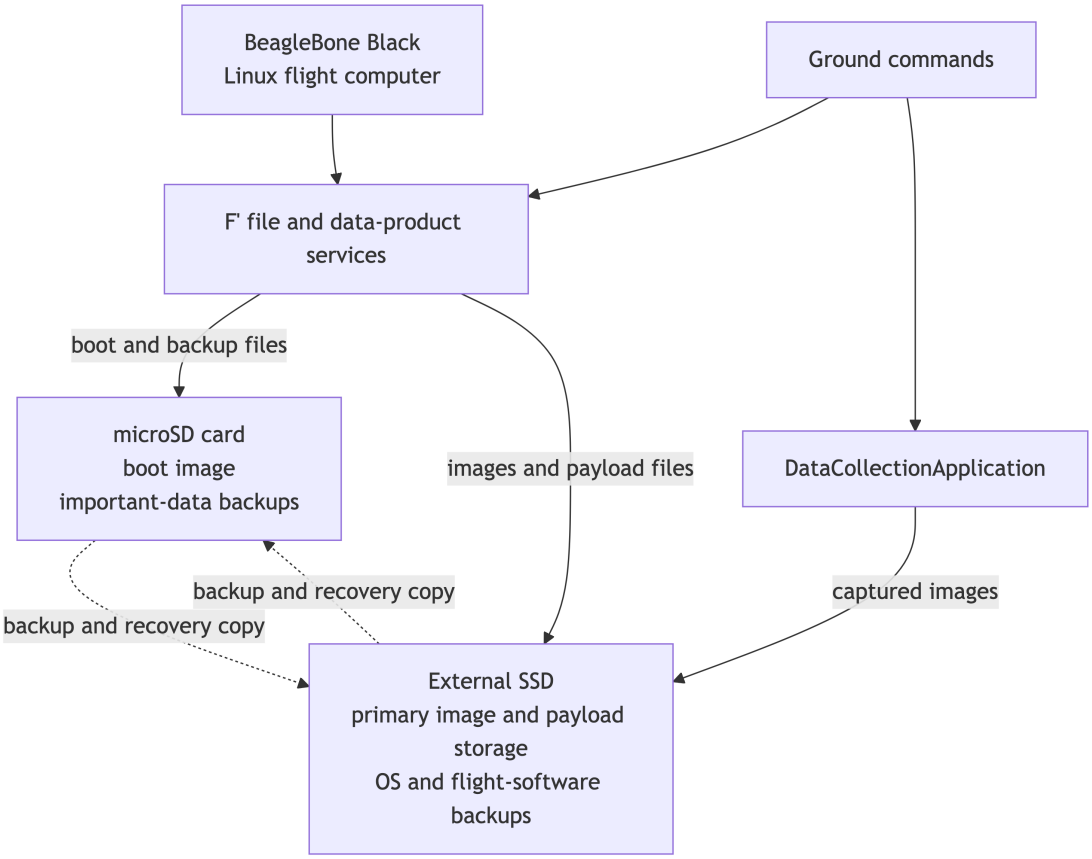
The original concept is documented in the archived FlightComputerMemory design.

29.2 Current Storage Architecture

The current design is intentionally simpler. The flight computer uses two nonvolatile storage devices:

- A microSD card is the boot medium. It is mechanically secured with epoxy and a structural latch to reduce the chance of connector movement or loss during launch and operation. It contains the bootable Linux and flight-software image, along with backups of important spacecraft data.
- An external SSD is the primary high-capacity storage device for mission data, especially captured images and other payload files. It also contains backups of the operating system and flight-software image so that a known-good copy is available even when the boot medium is being recovered or replaced.

The previous plan to boot or operate storage through the BeagleBone Black eMMC pins was removed because the required boot configuration could not be supported reliably on the selected hardware. The current design therefore does not depend on NOR flash, an external eMMC device, or the BeagleBone Black eMMC pins for normal booting.



File operations use the F' file-management (OSAL) and file-downlink services, while applications own mission-specific decisions such as when to capture images, which products to preserve, and when a file may be deleted. The final partition names, capacity margins, and file-system recovery procedure remain configuration items for the deployment and data budget.

29.3 Storage Partitioning Policy

The storage layout is divided by function so that a full payload area or a corrupted data file cannot prevent the flight computer from booting or recovering. The partition names and exact sizes are deployment configuration items, but the following logical arrangement is the baseline policy:

- **microSD boot/system region:** contains the boot configuration, Linux kernel, root file system, deployment configuration, and the normal flight-software image. It is treated as read-mostly during operations and is updated only through a controlled maintenance procedure.
- **microSD persistent-data region:** contains backups of important spacecraft data, including selected images for later downlink, persistent parameters, and fault/event logs. This region is separate from the boot/system region so routine data growth cannot consume the boot partition.
- **SSD payload region:** is the primary location for captured images, science results, and other large mission products. `DataCollectionApplication` writes new images here after capture and the science applications add processing results as they become available.
- **SSD recovery region:** contains the backup operating-system and flight-software images, together with the metadata needed to identify a known-good copy. It is not used as an unbounded payload directory.
- **SSD log and staging region:** contains event logs, temporary image files, compression outputs, and other files that are not yet committed as valid products. Its quota prevents diagnostic growth from consuming the entire payload region.

Both devices maintain a reserved free-space margin. The application shall reject or defer a new image capture before that margin is consumed, and log the reason for the rejection. Log rollover and backup selection shall also respect the margin. Exact thresholds will be set from the data budget and tested under full-storage conditions.

29.4 Storage Recovery Policy

The microSD card is the only normal boot medium. The BeagleBone Black eMMC pins and the retired NOR/eMMC architecture are not part of the normal boot path. The SSD recovery region provides a known-good software copy for a ground-directed maintenance or recovery operation; automatic boot fallback from the SSD is not assumed unless it is explicitly supported by the final deployment.

At startup and after a reset, the deployment shall apply this sequence:

1. Boot the Linux and flight-software image from the microSD boot/system region.
2. Mount the persistent-data region and SSD regions independently, and record an event for each unavailable or failed mount.
3. Validate the CRC and version/validity metadata for the software image, persistent parameters, logs, and any data product opened for use.
4. Ignore, quarantine, or replace any file that fails validation. When a valid backup exists on the other device, restore or copy it only through a controlled recovery operation and record the source and result.
5. Reconstruct runtime state from the valid persistent files. Data that existed only in RAM, including queued commands, unsent packets, and incomplete processing buffers, is considered lost after the reset.
6. Start nonessential payload activity only when the required storage region is available and has passed its checks. If the SSD is unavailable, the spacecraft remains able to boot and can continue essential operations from the microSD card while image capture and other storage-heavy work is disabled or deferred.

Writes shall use a temporary or in-progress filename and become visible as valid products only after the contents, metadata, and CRC have been written successfully. Startup shall quarantine in-progress files rather than treating them as complete images or science products. Image deletion shall be ground-commanded and shall occur only after the corresponding downlink or recovery operation has been acknowledged; the backup copy is retained until the deletion policy allows it to be removed.

The recovery procedure shall prioritize, in order, a bootable flight-software image, persistent configuration and parameters, fatal and fault logs, data needed to explain the fault, and selected images or science products. Every recovery action shall produce an F' event and, when possible, update the onboard log for later downlink. The final CRC format, backup cadence, and operator commands will be verified in GDS, subsystem, and integrated deployment tests.

29.5 Runtime RAM Contents

RAM is volatile and is not treated as a persistent archive. During normal operation it contains the working state required by the F' deployment and the mission components, including:

- F' component instances, object state, parameter caches, state-machine state, and health-monitoring counters;
- active-component thread stacks, queues, synchronization objects, and scheduler state provided through the F' OSAL;
- `Fw::Buffer` instances and buffers used by command, event, telemetry, file, and CCSDS packet paths;
- pending commands, event records, telemetry values, packetizer work, and communications queues;
- sensor samples, cached hardware-manager state, algorithm intermediates, and control-loop data; and
- temporary image, DMA, compression, and science-processing buffers.

The exact allocation depends on the deployment configuration and image-processing workload. Any state that must survive a reboot is written to the microSD card or SSD through the applicable F' file or data-product path. A reset will discard in-flight commands, queued packets, unsaved sensor samples, and incomplete processing buffers; software will complete writes or mark incomplete products when practical so that they can be detected during recovery.

29.6 Software-Level Radiation and Fault Mitigation

The spacecraft does not rely on the storage devices or RAM being radiation-hardened. The software-level approach is to detect corrupted or implausible state, preserve recoverable copies, and return the affected subsystem to a known state. Planned controls include:

- **Integrity checks:** persistent files and important data records use CRCs, with version or validity metadata where needed. A failed check causes the copy to be rejected rather than used as authoritative state.
- **Redundant copies:** the operating system and flight-software image are kept on both the SSD and microSD card. Important mission data, including image products selected for downlink, is backed up across the two devices when capacity and timing permit.
- **Atomic and recoverable file operations:** writes use temporary or otherwise clearly marked incomplete records until the final data is committed. Startup code and file services identify incomplete products and prevent them from being mistaken for valid images or state.
- **Input validation and state-machine recovery:** hardware managers validate register values and sensor ranges, use timeouts and retries for device transactions, and return to their reset or safe state after unrecoverable errors. Applications similarly reject invalid data before it reaches control or science algorithms.
- **Liveness monitoring and watchdog recovery:** `Svc::Health` detects missed component responses, while `WatchdogPinger` services the EPS hardware watchdog. Fatal conditions are reported through `Svc::EventManager` and `Svc::FatalHandler`; the application-manager interfaces can then apply the appropriate subsystem reset policy.
- **Safe operational behavior:** after a detected fault, the satellite discards the output, preserves fault context, and `Svc::FatalHandler` decides whether to reset hardware (if it is a more contained issue) or reset the system (if it is a larger issue such as flight software failing to run).

These mechanisms mitigate many software-visible effects of single-event upsets and corrupted storage, and we use both the microSD and SSD for redundancy purposes.

29.7 Computational Load and Power Protection

Image processing, compression, encryption, and ADCS filtering can create CPU, memory, storage-I/O, and power demands on the BeagleBone Black. Representative payload processing, image compression, encryption, and ADCS-filtering workloads have been run successfully on the target-class computer without observed overload. The software will still treat these operations as nonessential work and will protect the timing of command handling, health monitoring, EPS monitoring, communications, and ADCS control.

The deployment will likely [TBR] migrate from Debian Linux to Buildroot Linux. Buildroot will provide a smaller and more lightweight operating-system image, reducing the storage required for the redundant OS and flight-software backups while reducing unnecessary runtime services. If measurements show that standard Linux scheduling does not provide sufficient timing predictability, the deployment will apply the PREEMPT-RT patch to improve scheduler behavior for the flight-software and payload workloads.

The primary operational mitigation will be scheduling separation. The spacecraft will not start a new experiment while it is in the `Downlink` mode. Payload capture and processing will therefore not be scheduled in the same operational interval as the communications workload that transmits stored products. The state machine and application mode interfaces will enforce this separation; the applications will also reject or defer a new experiment if a conflicting downlink or processing operation is already active. Long-running processing, compression, and encryption will run in active or queued execution contexts rather than in synchronous command or control handlers.

If runtime measurements show insufficient CPU time, memory, storage-I/O capacity, voltage margin, or thermal margin, the flight software will reduce the experiment rate: it will accept fewer `RUN_EXPERIMENT` opportunities, process one image or algorithm chain at a time, defer compression or encryption, or pause nonessential payload work until the next allowable window. Pending work will remain in a recoverable state where possible, and the application will emit an event describing the deferral or interruption. The payload rate will be the adjustable resource; health, EPS protection, command handling, communications, and safety-critical control schedules will retain priority.

Low-voltage, brownout, thermal, watchdog, or liveness indications will immediately take priority over payload work. `SatStateMachine` can move the spacecraft to a safe or charge-oriented mode, and the application will quiesce its own work in response to that mode change. `Svc::Health`, the EPS watchdog, telemetry, and event logs will provide evidence of the interruption and resulting recovery. Final execution-time, queue-depth, buffer, voltage, and thermal thresholds will be set from the power and data budgets and verified in GDS, subsystem, and integrated deployment tests with representative processing loads.

29.8 Data Retention and Ground Operations

Continuous retention of every raw telemetry sample is not required. `Svc::TlmPacketizer` collects the configured telemetry channels for downlink, and between ground passes the flight software will use aggregates over approximately five-minute intervals rather than retaining every raw sample indefinitely. However, during a ground pass, we do also downlink real-time telemetry. The detailed volume and rates are described in the data budget.

`Svc::EventManager` packages F' events for downlink. Fatal events and other selected events are also written to an onboard log for ground review, with the log transferred through the file-downlink path. Logs will retain enough context to identify the component, fault class, and recovery action, subject to the final capacity and rollover policy.

Ground operations control the persistence and disposal of large products. Ground commands will be used to:

- request image downlink and select the images or files to transfer;
- delete images after successful downlink or when storage margins require reclamation; and
- schedule experiments, which determines when new images and associated science products are created.

The data budget defines the expected image volume, log retention, telemetry aggregation, downlink priority, and storage-margin calculations in more detail. This SDD defines the software ownership and recovery behavior.

30 Verification and Test Plan

Verification proceeds outward from deterministic component behavior toward hardware and system behavior. Every component has a test scope appropriate to its type, interfaces, state machine, and failure modes.

30.1 Component-Level Tests

Unit tests cover algorithms, port handlers, state transitions, parameter handling, command validation, telemetry publication, event severity, and error paths. Passive components will be tested synchronously with controlled inputs. Active and queued components will be tested for queue behavior, scheduling assumptions, and correct responses to health pings and asynchronous completion.

Driver tests will verify configuration, open/close behavior, valid and invalid bus operations, operating-system error translation, and safe behavior after a failed transaction. Hardware managers will be tested with simulated driver responses for initialization, register transactions, timeouts, retries, invalid data, recovery, and escalation to application-visible faults.

30.2 GDS Deployment Tests

Hardware managers and Linux drivers will be exercised in F' Ground Data System (GDS) deployments before flight hardware is available or integrated. GDS tests will send commands, inspect telemetry and events, verify parameter loading, and exercise the same typed ports used in the flight topology.

The GDS environment provides representative device interfaces or deterministic simulators for UART, I2C, SPI, GPIO, and PWM behavior. Tests include nominal responses, delayed responses, malformed data, bus failures, and device reset conditions. This allows recovery behavior to be verified without depending on nondeterministic hardware timing.

30.3 Subsystem Tests

Each subsystem will then be tested in an increasingly complete deployment:

1. connect the application to simulated managers and verify application state machines and algorithm selection;
2. connect managers to simulated or representative drivers and verify device-level data flow;
3. enable rate groups, health monitoring, commands, events, and telemetry together;
4. verify subsystem recovery behavior and safe outputs after manager or driver faults; and
5. verify the subsystem's communications, data-product, and storage interactions where applicable.

Examples include checking that ADCS algorithms produce safe actuator commands from known sensor inputs, that communications mode changes alter telemetry packet behavior, that EPS managers preserve safe power states after failures, and that thermal control does not enable a heater on invalid sensor data.

30.4 Integrated Topology Tests

Integrated tests will exercise cross-subsystem data flow, shared CDH services, CCSDS uplink/downlink paths, file operations, data products, rate-group scheduling, health monitoring, and the `SatStateMachine` mode transitions. Fatal-event routing and the application-owned targeted-reset path will be added to the integrated fault-response tests as those interfaces are finalized.

30.5 Verification Evidence

Each requirement will identify its verification method, such as inspection, unit test, GDS deployment test, subsystem test, or integrated test. Test results will retain the command sequence, configuration, input data, telemetry/events observed, expected result, actual result, and any hardware or simulator version needed to reproduce the run.

30.6 Related Verification Artifacts

This SDD defines the verification strategy and the component-level verification methods shown in the requirements tables. The following controlled documents provide the detailed operational and test evidence rather than duplicating it here:

Document	Relationship to this SDD
Command and Telemetry List (CTL)	Defines the operator-visible commands, telemetry channels, events, and command-response expectations used by the ground interface.
Ground Data System (GDS) document	Defines the ground hardware, ground software, and operator workflows used to send commands and observe flight responses during test and operations.
Schedule document	Defines the authoritative development milestones, review dates, and deadlines; this SDD does not duplicate those dates.
Requirements Verification Matrix (RVM)	Traces mission and system requirements to their verification methods, procedures, evidence, and closure status.
Unit, GDS, subsystem, and integrated test procedures and reports	Define execution steps, pass/fail criteria, configurations, as-run results, and anomaly records for the verification levels described in this SDD.
Data budget	Defines telemetry, event, image, storage, and downlink volume assumptions used to set the test configurations and acceptance criteria.

The source repository and test records will retain versioned procedures and results. Each verification record will identify the applicable SDD component and requirement IDs so that design, tests, and requirements remain traceable.

31 Software Development Process

HuskySat-2 flight-software source, deployment configuration, component design documents, and test artifacts are maintained as version-controlled engineering data. The process below applies the same configuration-management discipline to flight software that is used for hardware and system design.

31.1 Configuration Management

The team maintains flight-software and SDD changes in Git repositories. Each change records its author, review history, and associated issue or task. Released deployment configurations identify the source revision, generated artifacts, build environment, target hardware configuration, and test evidence so a result can be reproduced.

Changes that affect commands, telemetry, events, interfaces, or operational behavior will update the applicable SDD page, Command and Telemetry List (CTL), interface documentation, and verification records in the same change set or linked task. The team will retain prior test procedures and as-run results rather than overwriting historical evidence.

31.2 Component Design and Coding Standards

Each mission component will be developed from its SDD entry. The author will first define the component's ports, commands, events, telemetry, parameters, state-machine states, and failure responses in FPP, then update the SDD when the interface or behavior changes. FPP will remain the source of truth for the machine-checked interface; generated `*Ac.hpp` and `*Ac.cpp` files will not be edited by hand. Mission behavior will be implemented in the hand-written component `.hpp` and `.cpp` files and will be covered by the component's tests.

The C++ implementation will follow these standards:

- Names will identify the subsystem and purpose of a function, variable, port, event, or telemetry channel; unexplained abbreviations and magic numeric constants will not be used.
- Comments will explain intent, assumptions, units, state-transition rationale, timing constraints, ownership, and fault handling. Comments will not merely repeat what an obvious line of code does. Non-obvious algorithms, hardware workarounds, and recovery paths will include a short explanation and a reference to the applicable requirement or design decision.
- Every command handler will validate its arguments and current operating state before changing hardware or persistent data. It will return a defined command status and emit the appropriate event for rejection, timeout, retry exhaustion, or other abnormal behavior.
- Every asynchronous or queued path will document buffer ownership, queue behavior, timeout behavior, and what happens when the queue or storage is full. Shared state will have an explicit owner and synchronization strategy.
- Initialization, normal operation, reset/recovery, and shutdown behavior will be visible in the component state machine or design notes. Hardware managers will leave devices in a known safe state after failed initialization or unrecoverable communication errors.

31.3 Branching, Review, and Merge

Each change will use a scoped Git branch tied to a task, issue, or design decision. A pull request will describe the reason for the change, affected components and interfaces, test commands and results, and any remaining limitations. Direct changes to the protected integration branch will not be used for normal development.

Before requesting review, the author will build the affected deployment, run the relevant unit tests, check formatting and generated-code consistency, and update the SDD, FPP, CTL, and requirements traceability when applicable. A reviewer will verify:

- the implementation matches the FPP interface and SDD behavior;
- command validation, event severity, telemetry, timeouts, retries, and fault recovery are defined;
- active-component threading, queue sizes, buffer ownership, and rate-group timing are safe;
- hardware access remains inside the appropriate manager or driver boundary;
- tests cover normal operation, boundary values, invalid inputs, and failure paths; and
- comments, names, documentation, and repository changes are clear enough for another team member to maintain.

Changes to `SatStateMachine`, `Svc::FatalHandler`, storage recovery, command routing, shared services, or cross-subtopology wiring will receive review from both the responsible subsystem owner and the flight-software integration owner. Merge conflicts will be resolved by the author, followed by a fresh build and test run before approval. A pull request will not be merged when required tests fail, the SDD and interface model disagree, or generated files are modified without the source FPP change.

31.4 Test, Release, and Issue Tracking

The verification sequence is component test, GDS deployment test, subsystem test, and integrated-topology test, as defined in the Verification and Test Plan appendix. Component tests will exercise state transitions, command acceptance and rejection, boundary values, invalid sensor or bus data, timeout and retry behavior, queue/storage exhaustion, and fatal or recovery paths where applicable. GDS tests will verify topology wiring, command responses, event and telemetry visibility, rate-group behavior, and deployment startup. Subsystem and integrated tests will use representative hardware and workload configurations.

Test procedures and reports will identify the source revision, FPP/generated-code state, build configuration, hardware or GDS deployment, inputs, pass/fail criteria, observed telemetry and events, anomalies, and as-run result. A release candidate will be tagged or otherwise identified only after the required evidence for its intended test level has been reviewed. Test failures and anomalies will be recorded as issues with an owner, reproduction information, severity, disposition, and regression-test status.

The team will track work in the program schedule and task tracker. Each task will identify an owner, dependencies, planned milestone, and completion criteria. The Schedule document is the authoritative source for milestone dates, deadlines, and review dates; this SDD will refer to that document rather than duplicate dates that can become stale. Milestones will cover component completion, hardware-manager integration, application/subsystem deployment, integrated flight-software deployment, and the test-readiness points needed for the UNP review schedule.

31.5 Roles and Accountability

The following role definitions establish the responsibilities that will be assigned to named team members in the controlled team roster before final CDR submission:

Role	Responsibility
Flight Software Lead	Owns software architecture, integration decisions, review readiness, and overall delivery of flight-software evidence.
Subsystem Software Owner	Owns the subsystem application's and managers' requirements, design, implementation, unit tests, and documentation.
Flight Software Integration and Test Owner	Owns deployment configuration, GDS/subsystem/integrated test planning, test evidence, and anomaly tracking.
Configuration and Release Owner	Owns release identification, build reproducibility, repository hygiene, and configuration records.
Ground Interface Owner	Coordinates the CTL, GDS interface, command-response behavior, telemetry/event visibility, and operational test interfaces.

32 Revision History and Update Log

This appendix summarizes the SDD-relevant development history of the `hs2-software-design` repository. Merge commits and commit identifiers are omitted; the Git repository remains authoritative for the complete commit record, review history, and file-level changes.

The project history identifies an initial LaTeX version of the SDD authored by **Josh Lando** in **August 2025**.

Period	Contributors	Grouped SDD-relevant work
August 2025	Josh Lando	Initial HuskySat-2 Software Design Document in LaTeX.
April 2026	Josh Lando; Piyush Acharya	Established the repository and initial SDD structure; documented the mission architecture, hardware-manager startup assumptions, operational modes, architecture status, and draft ADCS, Data Collection, and Science Inference managers.
May 2026	Josh Lando; Henry Adams	Added and reorganized EPS, ADCS, thermal, communications, driver, panel-deployment, watchdog, and hardware-manager documentation; added requirements, recovery behavior, and CCSDS topology updates.
May 2026	Ojeet Deol	Began the communications application documentation.
July–August 2026	Senuka Liyanage; Eleanor Marso; Ojeet Deol; Henry Adams	Expanded thermal, GNSS, sun-sensor, EPS, and communications designs; added component SDDs, telemetry and event behavior, state-machine changes, diagrams, and document clean-up.
August 2026	Senuka Liyanage; Eleanor Marso; Ojeet Deol; Henry Adams	Continued component review and refinement, including heater controls, manager naming, communications modes, downlink behavior, requirements, and topology diagrams.
September 1–8, 2026	Eleanor Marso; Ojeet Deol; Mahir Emran	Added the SatStateMachine SDD, updated radio and SDLS documentation, incorporated meeting feedback, and reworked SDD diagrams.
September 9, 2026	Senuka Liyanage; Mahir Emran; Eleanor Marso	Updated MagnetorquerManager, ImmuManager, CameraManager, DataCollectionApplication, GNSS, sun-sensor, and Linux PWM driver documentation; removed the obsolete Linux USB driver page.
September 10, 2026	Senuka Liyanage; Eleanor Marso; Ojeet Deol; Mahir Emran	Reorganized the SDD, resolved naming and diagram conflicts, aligned ADCS and SatStateMachine interfaces, documented communications events, and updated subsystem modes and application behavior.
September 11–15, 2026	Senuka Liyanage; Eleanor Marso; Mahir Emran	Integrated the latest Science Inference, Thermal, Camera, StarTracker, SatStateMachine, and ADCS application updates, including state diagrams, table fixes, Kalman-filter material, and the ADCS topology diagram.
September 16, 2026	Senuka Liyanage; Mahir Emran	Integrated the latest Data Collection, Camera Manager, and Science Application changes from `main`; standardized the component name as `ScienceApplication`, resolved the file-name conflict, and aligned the merged content with the current SDD structure.

Period	Contributors	Grouped SDD-relevant work
September 16, 2026 (diagram update)	Mahir Emran	Replaced the high-level, payload, and thermal topology figures; added communications and EPS topology figures; documented PAY/ADCS ActiveRateGroup phase offsets; and clarified the 10 Hz, 1 Hz, and 0.1 Hz rate-group assignments.

The current PDF build also includes document-structure, formatting, and compilation fixes maintained on the SDD build branch.

33 Acronyms

Acronym	Meaning
ADCS	Attitude Determination and Control Subsystem
AI&T	Assembly, Integration, and Test
CCSDS	Consultative Committee for Space Data Systems
CDR	Critical Design Review
CDH	Command and Data Handling
CRC	Cyclic Redundancy Check
CTL	Command and Telemetry List
eMMC	Embedded MultiMediaCard
EPS	Electrical Power Subsystem
F'	F Prime flight-software framework
FDIR	Fault Detection, Isolation, and Recovery
FPP	F Prime Prime modeling language
GDS	Ground Data System
GPIO	General-Purpose Input/Output
GNSS	Global Navigation Satellite System
I2C	Inter-Integrated Circuit
IMU	Inertial Measurement Unit
NOR	Not OR flash memory
OSAL	Operating-System Abstraction Layer
PID	Proportional-Integral-Derivative
PPS	Pulse Per Second
PWM	Pulse-Width Modulation
RAM	Random-Access Memory
RVM	Requirements Verification Matrix
SDD	Software Design Document
SPI	Serial Peripheral Interface
SSD	Solid-State Drive
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus

34 References

This section collects the external documentation and design sources referenced by the SDD. Component pages describe how each source applies to HuskySat-2; this list provides the direct links in one place.

34.1 F' Framework and FPP

- F' ports, components, and topologies
- F' commands, events, channels, and parameters
- FPP state-machine guide
- FPP state-machine inherited transitions
- FPP state-machine substates
- F' rate-group pattern
- F' health-checking pattern
- F' manager-worker pattern
- F' application-manager-driver pattern
- F' subtopologies pattern
- F' common port patterns
- F' ground interface
- F' data products
- F' building a topology
- CdhCore subtopology definition
- ComCcsds subtopology definition
- FileHandling subtopology definition
- DataProducts subtopology definition
- F' reference deployment instances
- F' reference deployment topology
- Svc::Health FPP definition

34.2 F' Drivers and OSAL

- Drv::LinuxGpioDriver
- Drv::LinuxI2cDriver
- Drv::LinuxPwmDriver
- Drv::LinuxSpiDriver
- Drv::LinuxUartDriver
- F' OSAL source

34.3 F' Services and Components

- Svc::ActiveRateGroup
- Svc::AssertFatalAdapter
- Svc::Ccsds::ApidManager
- Svc::Ccsds::SpacePacketDeframer
- Svc::Ccsds::SpacePacketFramer
- Svc::Ccsds::TcDeframer
- Svc::Ccsds::TmFramer

- Svc::CmdDispatcher
- Svc::ComAggregator
- Svc::ComQueue
- Svc::DpCatalog
- Svc::DpManager
- Svc::DpWriter
- Svc::EventManager
- Svc::FatalHandler
- Svc::FileDownlink
- Svc::FileManager
- Svc::FileUplink
- Svc::FprimeDeframer
- Svc::FprimeFramer
- Svc::FprimeRouter
- Svc::FrameAccumulator
- Svc::Health
- Svc::PrmDb
- Svc::RateGroupDriver
- Svc::TlmChan
- Svc::TlmPacketizer
- Svc::Version

34.4 HuskySat-2 and Mission References

- HuskySat-2 software-design repository
- UWCubeSat GitHub organization
- Archived FlightComputerMemory design
- fprime-community sensor component reference
- F' Manager-Worker example
- F' Worker example
- F' Manager-Worker subtopology example
- F' Manager component definition
- F' Worker component definition
- F' Manager-Worker subtopology definition
- F' ByteStreamDriverModel reference