Software Design Document

HuskySat-2

Husky Satellite Lab
University of Washington

Contents

1 Overview

Software in the HuskySat-2 system consists of flight software (FS) and ground software (GCDH). The command and data handling team (CDH) is in charge of maintaining this document with help from the ground software team. The CDH team is responsible for providing common storage and a framework to integrate the code from each subsystem in the flight software. The flight software will be implemented in F Prime [TBR]. Ground software will receive, store, and process data received from the satellite. Ground software will also provide tools to command the satellite. Ground software will be implemented with [TBD] software.

The software block diagrams use the C4 model which consists of three levels: system, container (NOT related to Docker), and component. System is the highest level of abstraction and describes how the software delivers value (science data) to our customers (the science team). Container is the second level of abstraction and describes (mostly) self-contained processes. These processes are run independent of one another. For HuskySat-2, most container level blocks represent software for a specific subsystem. That subsystem is responsible for writing that code [TBR]. Component level diagrams are the final level of abstraction, and are one layer above code/implementation, which we have chosen not to draw block diagrams for. In object oriented programming, a component is a group of related implementations of an interface.

NOTE: All systems/containers/components are software unless specified otherwise.

NOTE TO PDR REVIEWERS: Due to timing constraints, parts of this document will be incomplete – an updated pdf of the software design document can be found in our github.

1.1 HuskySat-2 System

The HuskySat-2 Software system has four different categories: hardware, database, software, and people. The following diagram (Fig 1.1) shows how the ground and science team use the ground software to communicate with the satellite. The ground software and flight software talk to each other through the [TBD] ground station and [TBD] communications hardware, respectively. The ground station is responsible for visualizing the satellite state of health and progress of the science mission. The telemetry and visuals provided by the ground software will be analyzed by the ground and science team, who will then use the ground software to send commands to the satellite. The ground software will store a history of all operation and science data in a database [TBD]. The flight software is written in F Prime [TBR]. Flight software is responsible for executing commands from the ground station as well as autonomously maintaining the health of the satellite. The flight software is in charge of collecting

all science data and running LOST and FOUND. The command data handling team is responsible for integrating flight software with the satellite hardware [TBR].

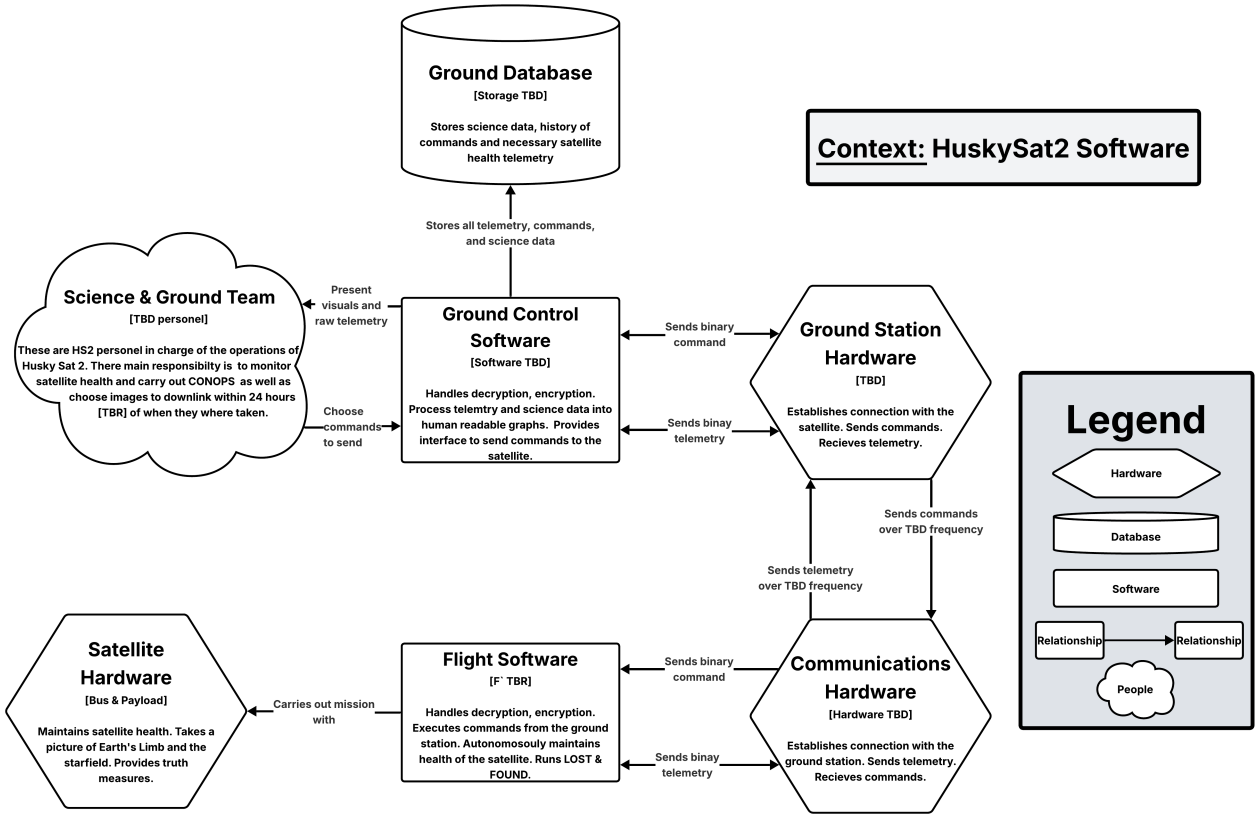

Fig. 1.1: The HuskySat2 software system

1.2 Flight Software

The flight software for HS-2 serves as the central nervous system of the satellite, enabling autonomous attitude control and mission execution in Low Earth Orbit. This section outlines the architecture, functional requirements, and verification strategies for the flight software. The flight computer is currently a Beaglebone Black (1GHz ARM Cortex-A8, 512 MB RAM). While compute requirements for FOUND have not been concretely established, the entire LOST pipeline has been run on comparable hardware (Raspberry Pi 4) in less than 50 ms.

1.2.1 Requirements

The Command and Data Handling (CDH) subsystem for HS-2 is designed to serve as the brain of the satellite. Key requirements include the ability to process and store experiment data, execute attitude control and thermal control algorithms, handle telemetry downlink, and provide fault tolerance.

ID	Requirement
CDH-1	The flight computer shall implement the state machine for the satellite's operational modes, as specified in the Mission Design Document.
CDH-2	The flight computer shall provide at least 25 GB [TBR] of non-volatile storage for attitude data and timestamped images from LOST and attitude truth measure.
CDH-3	The flight computer shall support a minimum receiving bandwidth of 500KBps [TBR] for attitude data and timestamped images from LOST and attitude truth measure.
CDH-4	The flight computer shall be capable of storing at least 31 GB [TBR] of non-volatile storage for position vectors and timestamped image data from FOUND and position truth measure.
CDH-5	The flight computer shall support a minimum receiving bandwidth of 625KBps [TBR] for position vectors and timestamped image data from FOUND and position truth measure.
CDH-6	The flight computer shall send commands to subsystems including ADCS, EPS, and THR, as defined in the ICD.
CDH-7	The flight computer shall package all telemetry as per the CTL.
CDH-8	The flight computer shall forward all telemetry to the communications system at minimum 50 kbps [TBR] during ground station passes.
CDH-9	The flight computer shall calculate and monitor Health Metrics as per UNP12-68 and UNP12-69.
CDH-10	The flight computer shall detect and respond to subsystem faults, as outlined by the fault management procedures document [TBD].
CDH-11	The flight computer shall run the ADCS control loop at 5 Hz [TBR, can be much more].

Tab. 1.1: Flight Computer Requirements

The payload software requirements for HuskySat-2 are tailored to enable successful operation and validation of LOST and FOUND. The payload software places requirements on the clock rate and RAM required for successful LOST and FOUND operation, as defined by the customer.

ID	Requirement
PAY-9	The payload shall determine pointing knowledge to within 20 [TBR] arcsec across LOST's boresight and 20 [TBR] arcsec around LOST's boresight.
PAY-10	LOST compute shall have a clock rate 100 MHz and 1 MiB of RAM.
PAY-11	FOUND compute shall have a clock rate [TBD] and [TBD] of RAM.
PAY-12	The payload shall determine the Earth Centered Inertial (ECI) coordinates to 1000 [TBR] m.
PAY-13	The payload shall be capable of determining the attitude of the satellite using a truth measure [TBD] times to an accuracy of 0.05 [TBR] degrees.

Tab. 1.2: Payload Compute Requirements

1.2.2 Block Diagrams

The flight software block diagram is a container level diagram in the C4 model (defined in the Overview). This diagram (Fig 1.2) is structured with a top-to-bottom flow down from communications to flight software to flight hardware. This shows the hierarchical chain of command in the satellite.

Communication

The top of the flight software diagram is the communications hardware. Two software modules connect to the communication hardware: communication receive and communication transmit. The communication receiver parses all binary from the communication module and decrypts it into commands which are stored in the flight software storage. The communication receiver is always on and is independent of the current state. The communication receiver is a single process and does not have a component level diagram. The communication transmitter is constantly downlinking an alive command. The communication transmitter is responsible for establishing a connection with a ground station during a pass. Depending on the state defined by the state machine, the communication transmitter will downlink satellite health telemetry and then science telemetry. For more information see Chapter 4.

Command Data Handling

The flight storage is organized into folders and files [TBR]. The flight storage is responsible for storing commands from the ground station, thermal telemetry, power telemetry, adcs telemetry, payload telemetry, and all error messages. The state machine is responsible for carrying out the commands from the ground station. The state machine is also responsible for choosing and enforcing a state (safe detumble, safe, standby, experiment) [TBR]. The payload storage is responsible for storing all science telemetry. It is also responsible for providing the most recent position and attitude measurements for other software components to access. For more information see Chapter 2.

Payload

The payload software is in charge of running the science experiments as well as interfacing with all the payload hardware. The payload software owns the star tracker and GNSS receiver. ADCS can request attitude and position information which it will be able to access in the payload storage. The payload software is responsible for making sure the requirements for an experiment are met before beginning. The payload software stores all science data in the payload storage. For more information see Chapter 3.

Attitude Determination Control

The attitude determination and control software (ADCS) is responsible for carrying out the pointing and slew rate commands from the state machine. It interfaces with attitude and position sensors as well

as data from payload storage to determine the state of the satellite. It then powers the magnetorquers to perform the desired attitude maneuver. For more information see Chapter 5.

Electrical Power System

The electrical power software is responsible for maintaining a healthy power st ate and reporting the battery capacity to the flight storage. The electric power system is responsible for interfacing with the electric power sensors and components hardware. The electric power system runs independent of the state of the satellite. For more information see Chapter 6.

Thermal

The thermal software dynamically powers the satellite's Kapton heaters based on readings from the temperature sensors. The software uses a Proportional-Integral-Derivative (PID) control algorithm to maintain the satellite components within their operating temperature ranges, and stores temperature data on the flight computer storage. It does not have a separate component diagram, since it is a single process. For more information see Chapter 7.

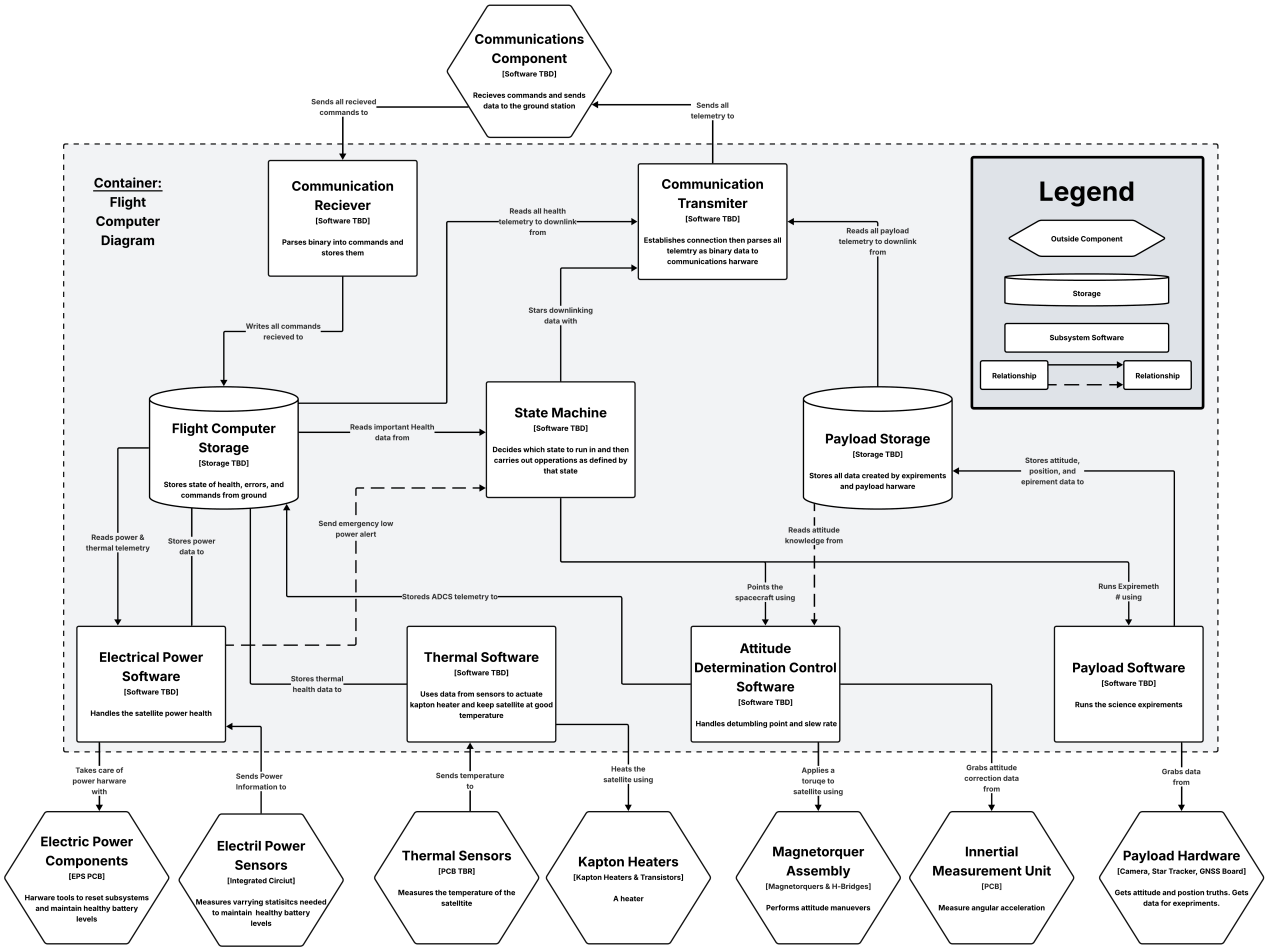

Fig. 1.2: The flight software diagram: container level

1.3 Ground Software

The ground software for HS-2, focused on the Ground Command and Data Handling (GCDH) system, enables efficient satellite communication, command uplink, telemetry downlink, and data analysis for the mission. The GCDH is designed to run on a ground station computer here at the University of Washington in Seattle with [TBD] GB RAM and [TBD] storage, interfacing with hardware for S-band operations. The ground software architecture is designed to support real time signal processing and validation.

1.3.1 Ground Software Architecture Overview

System Components: The GCDH includes the Command and Control UI for user interaction, storage manager for data handling, transmit manager for uplink encryption, receiver manager for downlink decryption, and satellite location propagation API for pass prediction. It uses the ground station's radio for S-band communications and a database for archiving.

Uplink: Command and Control UI creates commands, sends them to the Transmit Manager for encryption [TBR], which forwards transmission data to HackRF via Ethernet [TBR] .HackRF converts to S-band signal, amplified by the Power Amplifier [TBD], filtered by Band-Pass filter [TBD] and transmitted via Parabolic S-band antenna [TBR] to the satellite.

Downlink: S-band RF Waves from HS-2 are received by the Parabolic S-band Antenna [TBR], amplified by the Low-Noise Amplifier [TBD] to strengthen signals while minimizing noise, filtered by the Band-Pass Filter [TBD] to remove unwanted frequencies and converted to digital by HackRF, sending received data to Receive Manager via Ethernet [TBR].

1.3.2 Requirements

ID	Requirement
GCDH-1	The Ground Command and Data Handling system shall store a minimum of [TBD] GB of data.
GCDH-2	The Ground Command and Data Handling system shall have a user interface to input commands from the Command and Telemetry List (CTL).
GCDH-3	The Ground Command and Data Handling system shall provide visualizations of State of Health data from up to 48 [TBR] hours ago.
GCDH-4	The Ground Command and Data Handling system shall process and send telemetry to the science team as per the Command and Telemetry List.
GCDH-5	The Ground Command and Data Handling system shall be able to decrypt downlinked telemetry from the satellite.
GCDH-6	The Ground Command and Data Handling system shall be able to encrypt uplink commands.

Tab. 1.3: Ground Computer Requirements

1.3.3 Block Diagrams

Note to Reviewers: Ground software is nascent

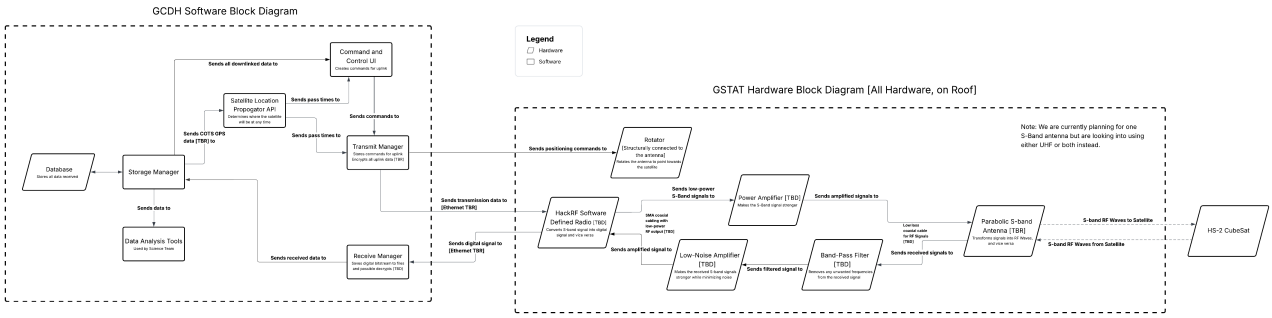

Fig. 1.3: Preliminary ground software

1.3.4 Ground Software Components

1. Command and Control UI

- Function: Creates and stores commands for uplink, interfacing with telemetry dictionary
- Inputs: User inputs from the ground team, pass times from satellite location propagator API

- Outputs: commands sent to Transmit Manager for encryption and transmission. Commands to rotator for antenna
- Interactions: stores commands in Storage Manager; coordinates with Transmit Manager for Ethernet data to HackRF
- Meets requirements: supports command generation with user-friendly interface (GCDH-2).

2. Transmit Manager

- Function: encrypts uplink data using TBD protocol and sends transmission data to HackRF for S-band conversion
- Inputs: Commands from Command and Control UI, pass times from Propagator API
- Outputs: Transmission data to HackRF via Ethernet [TBR], low power S-band signals to Power Amplifiers
- Interactions: Receives pass time for timing; interfaces with GSTAT hardware for amplification and antenna transmission
- Meets requirements: handles TBD kbps uplink, meeting the need for encrypted, timely command delivery during passes

3. Receive Manager

- Function: saves digital bitstream to files, decrypts [TBD], and sends received data to Storage Manager.
- Inputs: digital signals from HackRF (Ethernet, TBR), pass times from Propagator API
- Outputs: received data to storage manager for archiving
- Interactions: processes signals from Low-Noise Amplifier and Band-Pass Filter via HackRF; feeds decoded telemetry to analysis tools
- Meets requirements: enables decoding (TBD tool)

4. Storage Manager

- Function: Stores all received data in the database, manages archiving and retrieval
- Inputs: Received data from Receive Manager, TBD data from Propagator API

- outputs: Data sent to Data Analysis Tools [TBD] for science team use
- Interactions: Interfaces with database for storage
- Meets requirements: manages TBD storage for TBD MB/day telemetry

5. Satellite Location Propagator API
    - Function: Determines satellite position at any time using TLEs, sending pass times to managers
    - Inputs: Satellite TLEs
    - Outputs: Pass time to Transmit/Receive Managers, TBD data to storage manager
    - Interactions: Feeds positioning commands to Rotator [TBD] for antenna pointing
    - Meets requirements: Supports pass prediction for TBD hours daily coverage, enabling effective S-band operations
6. Data Analysis Tools
    - Function: Used by the science team for analysis of experiment telemetry
    - Inputs: data from storage manager
    - outputs: Reports and visulaizations for accuracy assessment [TBD]
    - Interactions: Interfaces with database for experiment data
    - Meets requirements: allows science team to determine LOST and FOUND's accuracy

Command Data Handling

2

2.1 State Machine

The CDH state machine is a finite state machine (FSM), as it defines distinct states with defined transitions. It is coded in [TBD] with an RTOS for real-time task scheduling. It uses a bitfield to represent subsystem states, where each bit corresponds to a subsystem's on/off status. States are triggered by internal events (e.g., EPS state-of-health via I2C) or external commands from ground. All state changes and subsystem statuses are logged to non-volatile flash (TBR) for post-event analysis. The FSM includes 4 satellite modes: Safe, Standby Eclipse, Standby Sunlit, and Experiment, with subsystem states encoded as binary flags in the software.

2.2.2.1 Mission Phases Tables

Legend: 0 = off, 1 = on

Tab. 2.1: Mission Phases Overview

PHASE 0: DEPLOYMENT	ADCS	CDH	PAY	EPS	Solar Panels	COMMS	THR
Launch	0	0	0	0	0	0	0
Deployment	0	0	0	1	0	0	0
PHASE 1: BUS COMMISSIONING	ADCS	CDH	PAY	EPS	Solar Panels	COMMS	THR
CDH Bootup	0	1	0	1	0	1	0
Detumble (MT)	1	1	0	1	0	1	0
Initial Charge	1	1	0	1	1	1	1
Downlink Status to Ground	1	1	0	1	1	1	0
PHASE 2: PAYLOAD COMMISSIONING	ADCS	CDH	PAY	EPS	Solar Panels	COMMS	THR
CO.1	1	1	0	1	1	1	1
CO.2	1	1	1	1	1	1	1
CO.3	1	1	1	1	1	1	1
CO.4	1	1	1	1	1	1	1
CO.5	1	1	1	1	1	1	1

2.1.1 State Transitions

Transitions are evaluated in a main RTOS loop every [TBD] seconds. Promotion (Safe->Standby, Standby -> Eclipse) requires passing all health checks, as outlined in the mode machine (Figure Xx). Demotion occurs immediately on anomalies like low battery. Autonomous logic includes eclipse detection via the ADCS sun sensors for Standby variants. Ground overrides authenticated (AES-TBD) and can force transitions. The subsystemStates bitfield is updated based on mode enetry/exit logic:

- Promotion: requires all health checks pass. subsystem flags set accordingly
- Demotion: Anomalies force safe mode, resetting affected subsystems to 0
- Autonomous overrides: eclipse detection via sun sensor toggles ADCS pointing mode; low SOC disables non-essential subsystems (payload)
- Manual control: Ground commands override states, authenticated via AES-TBD, updating subsystemStates directly.

In software, this is managed with a state enum [TBR, may be struct] and bitfield manipulation. For example, the state of ADCS is defined as a single bit as seen below:

```
enum State {INIT, DETUMBLE, SAFE, STANDBY, EXPERIMENT, DOWNLINK};
#define ADCS_BIT (1<<0) //Bit 0: ADCS on/off
```

To update subsystem states and propagate to hardware, the following functions are implemented:

There are also preliminary placeholder hardware interface functions:

A queue will handle events, such as low power or ground commands, with a [TBD] buffer to ensure real-time responsiveness. As shown in the code, bitwise operations manage subsystem states. There will be anomaly detection (FDIR TBR) to trigger safe mode and reset non-essential subsystems.

2.2 Flight Computer Storage

The file storage will be setup as follows.

```
/
├── commands/
├── experiments/
├── error-log/
├── power/
├── thermal/
└── adcs/
```

2.3 Payload Storage

The CDH subsystem is also responsible for managing payload data storage. This section details the storage architecture, leveraging the BeagleBone's onbaord eMMS storage and the potential integration of a microSD card to accomodate the high data volume of images from LOST and FOUND.

2.3.1 Storage Overview

The Beaglebone Black provides approximately 4GB of eMMC flash storage, suitable for storing telemetry, state logs, and processed payload data (quaternions, position vectors). However, raw image data from LOST and FOUND cameras, 1-2MB/image (TBR), with a target of 10-20 images per orbit (CONOPS) can quickly exceed this capacity over the mission duration. To address this, we have considered adding a microSD card to the BeagleBone, which would provide up to 32 GB of storage.

2.3.2 Storage Architecture

The CDH software implements a tiered storage strategy, managed by a RTOS task to ensure data integrity and availability.
Data Flow:

- The payload cameras capture images during Experiment mode, processed via LOST and FOUND algorithms to generate attitude and position estimates.
- The cameras we have currently selected compress the images by 50 percent, jpeg. These images are written to the microSD card, while processed data form the experiments is stored in the eMMC.
- During downlink, buffered data from the eMMC is prioritized for transmission via the EnduroSat radio, with images downlinked based on availabkle bandwidth.

File System:

- eMMC uses an ext4 file system for reliability, pre-allocated with a 100 MB circular buffer to handle power interruptions.
- MicroSD uses FAT32 for compatibility with ground station software, formatted with a 4 GB partition to balance speed and capacity.

Capacity Management:

- the CDH monitors storage usage, triggering a low space warning logged to eMMC
- if the microSD card fills, the oldest imgaes are overwritten in FIFO manner, ensuring continous operation until downlink

2.3.3 Storage Management

The storage management is integrated in the CDH State Machine, with a separate task handling I/O operations to avoid blocking the FSM. Key components include:

- Data writing
- Downlink Preparation in downlink, the CDH reads files from the microSD or eMMC based on priority (SOH >metadata >images), using a queue to manage transmission order
- Fault tolerance MicroSD errors trigger a fallback to eMMC and is logged for ground analysis. Other fault management procedures are still in development.

3 Payload

THIS IS SOLELY FOR COPY AND PASTING AT YOUR CONVENIENCE DELETE IN FINAL DRAFT

Listing 3.1: Image Scheduler OutPuts

```
1 /**
2 *
3 *
4 *
5 */
```

END OF CONVENIENCE

There are five different types of components in the payload software diagram: commercial software, storage, payload software, science software, and flight computer software. Commercial software is included with the hardware payload purchases and is described in their respective documentation. Storage lives on the flight computer. Payload software is written by the payload team. Science software is written by the LOST and FOUND team and is delivered as a static binary. Flight computer software is written by the flight software team.

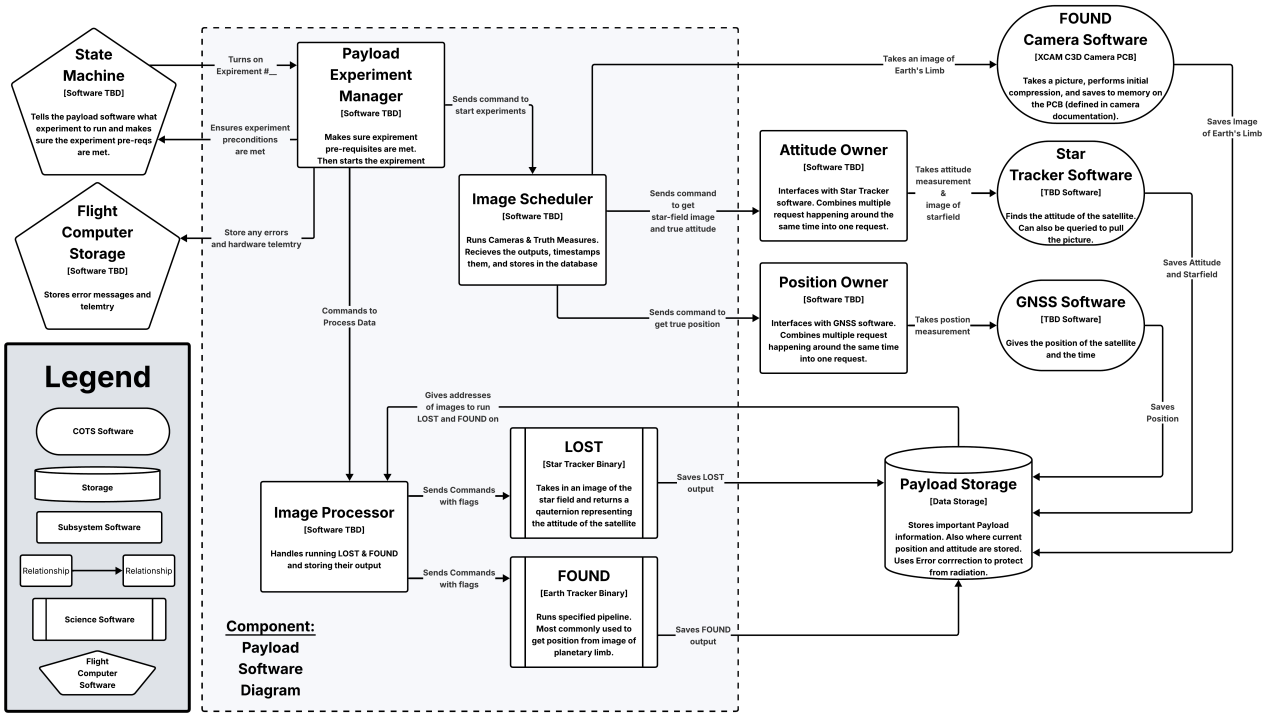

Fig. 3.1: The payload software: component level

3.1 The Payload Experiment Manager

The payload experiment manager is in charge of running the experiment and alternates between capturing images and processing the images with LOST and FOUND. The payload manager loads the experiment specified by the state machine into memory from the flight computer storage. It then runs the experiment.

Inputs

This object contains all of the information needed to run an experiment.

Listing 3.2: Experiment Struct

```
1
2  /**
3   * This is the code passed to the FlightComputerStorage to obtain the actual
        experiment
4   */
5      struct ExperimentCode {
6      int ExperimentCode;
7      }
8
9  /**
10  * contains all information on how an experiment should be run
11  */
12     struct Experiment {
13     /**
14      * Defines the experiment mode there are four experiment modes:
15      *
16      * 0: Run the star tracker and only get attitude information
17      * 1: Run the star tracker and get attitude information as well as a
18      *    picture of the starfield then process starfield with LOST
19      * 2: Run the FOUND camera and get attitude information from the star
20      *    tracker.
21      * 3: Run the FOUND camera and get attitude infromation from the star
22      *    tracker and LOST.
23      */
24     int experiment-mode
25     // number of experiments to run
26     int number
27     // number of time inbetween each experiment in seconds
28     int rate
29     }
```

The contents of the experiment and all of the optimizations occur within the state machine and flight computer managers. Payload will never change the contents of an experiment, only compartmentalize it to pass to the Utility Interface

Once an experiment is received the PEM will send preconditions back to the state machine for what attitude is required for the cameras to be able to take the proper image.

Outputs

Regardless of whether or not the preconditions are met the experiment manager will continue to send out outputs. Here are defined outputs that are for the fetching of the experiment.

Listing 3.3: Experiment Struct

```
1
2       PEMtoFlightComputer {
3       int Experiment Code;
4       }
5
6       PEMtoStateMachine {
7       bool arr[n]; //Confirms preconditions state, this is the attitude matrix
            required
8
9       }
```

Ports

Listing 3.4: Data Component Sends Out

```
1
2  /**
3  *    Takes in Experiment values to be send to the Attitude Owner
4  *    and Position Owner
5  */
6
7       PEM -> StateMachine (PORT)
8       output port PEMoutput: PEM -> StateMachine (PORT) //PEM Side
9       //Input port is defined by CDH, currently TBD
10
11      PEM -> FlightComputerStorage (FCS) (PORT)
12      output port PEMoutput: PEM -> FlightComputerStorage
13      outport port FCSoutput: FCS -> PEM
14      synch input port PEMInput: FCS -> PEM
15      //Input of FCS is defined by CDH
16
17
18 }
```

Preconditions, Post Conditions, Errors

For the FOUND/LOST along with our star tracker truth measure to function correctly the satellite must have the correct attitude. This is our Precondition. Currently there is no check to see if the satellite has the correct attitude and instead a command is sent through the flight computer (state machine) which then communicates with ADCS to adjust the attitude. The picture and adjustment occur asynchronously, as in simultaneously. The satellite should have attitude adjusted by the time of picture but will take the picture regardless of attitude. Before the command to ADCS is sent the PEM will check the current attitude.S

Implementation

Listing 3.5: Image Processor Implementation

```
1
2  active component PayloadExperimentManager {
3      void ExperimentIn_Handler (port num, context) {
4              Get Experiment
5              Split Experiment into 3 structs
6                  ->StarTracker
7                  ->FoundCam
8                  ->GNSS
9              Check utility softwares for active experiments
10                 if (false) {
11                     Send Experiments to Owners
12                 }
13                 else {
14                 Queue the experiments and wait for a rate group call to
                       recheck
15                     status of the utilities
16                 }
17     }
18
19 }
```

Testing

INSERT PSEUDOCODE

3.2 Image Scheduler (Within the PEM)

This is part of the PEM component as keeping a separate component is a bit redundant and introduces an unnecessary extra step. This takes advantage of the queued nature of an active component in FPrime. It will speak to the owners for what the frequency and amount of measurements to take

Outputs

The owners require an amount of measurements to take and a frequency to determine whether or not to turn off the utilities between measurements.

Listing 3.6: (PEM)Image Scheduler OutPuts

```
/**
*   This will establish how many photos and at what rate the star
*   tracker and GNSS receiver will be pulling attitude/position
*   measurements, these are sent at the same time.
*/
    struct StarTracker {
        int photos;
        float rateStar;

  }
   struct FoundCam {
        int photos;
        float rateFound;
 }

   struct GNSS {
        int positions;
        float rateGNSS;

   }

   }
```

Ports

A collection of port connection and input Handlers than depend on other port connections Ports can be repeated so different components have the same port (different handler implementations) The image scheduler worries about the current state of the utility software and the owners are just compartmentalizing the functions to keep talking to the utilities simple

Listing 3.7: Image Scheduler Ports

```

    //FOUND
     output sendToOwner(port num, context )
        -> synch input receiveCommand(context)
    //StarTracker
     output sendToOwner(port num, context )
        -> synch input receiveCommand(context)
    //GNSS Receiver
     output sendToOwner(port num, context )
```

```
10          -> synch input receiveCommand(context)
11
12      //Follows FIFO still
13       synch input SchedIn_handler: rategroup -> PEM(image scheduler)
```

Preconditions, Post Conditions, Errors

Preconditions: Cameras must be turned on and fully functional before the commands can be sent out to the owners.

Post Conditions: The last experiment has finished running before the the next experiment can be ran

Errors: -> Queue is empty = -1 Invalid Experiments = -2

Implementation

Listing 3.8: Image Scheduler Ports

```
1
2
3  active component PayloadExperimentManager {
4
5
6      schedIn_handler(port num, context) {
7          getStatus of current experiment //an enum?
8              -> if(null) {
9                  check if camera is on
10                     -> if(false) {
11                         send command to turn on
12                     }
13                     -> if(true) {
14                         popoff top of the queue
15                         send to handlers
16                     }
17
18             }
19             -> if(ongoing) {
20                 return 0; //does this even work here?
21             }
22             -> if(paused) {
23                 check if cameras are on
24                     -> if(false) {
25                         send command to turn on
26                     }
27                     -> if(true) {
28                         popoff top of the queue
```

```
                            send to handlers
                        }
                }


        }

}

}
```

Testing

TBD

3.3 Image Processor

Inputs

Listing 3.9: Image Processor Inputs

```
/** A command that allows the image processor to asynchronously process the image
*     that will be coming in decompress image for both FOUND and LOST + error  correction
*/
/**
    String flags: String representation of flags from PEM
    String filepath: address of image to run Lost & found on from payload storage

*/
struct ImageProcessorCommand {
    bool processWithLost;
    bool processWithFound;
    std::string flags;
    std::string filepath;
}
```

Outputs

Listing 3.10: Image Processor OutPuts

```
/**
* ProcessedImageData is not actually the output itself, just part of the command send to respective binaries
* LOSTCommand and FOUNDCommand contains imgPointer and flags.
*/

struct ProcessedImageData {
    uint32_t width;
    uint32_t height;
    float* intensityMatrix;
    size_t matrixSize;
    int processingStatus;
}

struct LOSTCommand {
    ProcessedImageData* LOSTimg;
    std::string flags;
}
struct FOUNDCommand {
    ProcessedImageData* FOUNDimg;
```

```
20      std::string flags;
21  }
22  int processingError;
23  /**
24  * Image file does not exist or is not accessible. processingError = -1
25  * Image format is invalid or unsupported. processingError = -2
26  * Unable to correct image errors. processingError = -3
27  * Cannot communicate with LOST/FOUND binaries. processingError = -4
28  */
```

Preconditions, Post Conditions, Errors

Preconditions: imageFilePath must point to a readable image file either processWithLost or processWithFound should be true Sufficient memory should be available for decompressed image matrix.

Post Conditions: Image is decoded, error corrected, and decompressed to intensity matrix Processed data is sent to specified binaries (LOST/FOUND) Original image file remains unchanged Resources (file handles, memory) are properly released

Errors: Image file does not exist or is not accessible. processingError = -1
Image format is invalid or unsupported. processingError = -2
Unable to correct image errors. processingError = -3
Cannot communicate with LOST/FOUND binaries. processingError = -4

Implementation

Listing 3.11: Image Processor Implementation

```
1   /**
2   * Takes an ImageProcessorCommand
3   * gets the compressed image from payload storage
4   * does error correction and decompression
5   * then forwards to LOST/FOUND
6   */
7
8   component ImageProcessor {
9       // called by PEM
10      void ProcessCommandIn_handler(const ImageProcessorCommand& cmd) {
11          processingError = 0
12          ProcessedImageData data;
13          data.processing.status = -1
14
15          // check that we're actually doing something
16          if (!cmd.process.WithLost && !cmd.processWithFound) {
17              processingError=-2;
```

```
18              return;
19          }
20
21          //This relies on the PEM having access to the storage, which needs
                to be added*
22          CompressedImage img = readCompressedImage(cmd.filepath);
23
24
25          if (!img.valid) {
26              processingError = -1;
27              return;
28          }
29
30          CorrectedImage corr = applyErrorCorrection(img);
31          if(!corr.valid) {
32              processingError = -3
33          }
34
35          // Error correction. We're going to go with error correction instead
                of detection to err on the side of safety. Since we don't know
                what the camera will see in orbit, we cannot determine the
                thresholds for error detection.
36
37          //img handling (meta)
38          data.width=corr.width;
39          data.height=corr.height;
40          data.intensityMatrix=decompressToIntensity(corr);
41          data.matrixSize=data.width*data.height;
42          data.processingStatus=0; //success
43
44          //LOST/FOUND
45          if(cmd.processWithLost){
46              LOSTCommand lc;
47              lc.LOSTImg = &data;
48              lc.flags=cmd.flags;
49              send_LOSTCommandOut(lc)
50          }
51          if(cmd.processWithLost){
52              FOUNDCommand fc;
53              fc.FOUNDImg = &data;
54              fc.flags=cmd.flags;
55              send_FOUNDCommandOut(lc)
56          }
57      }
58  }
```

Testing

Yet To Be Written

3.4 Utility Interface

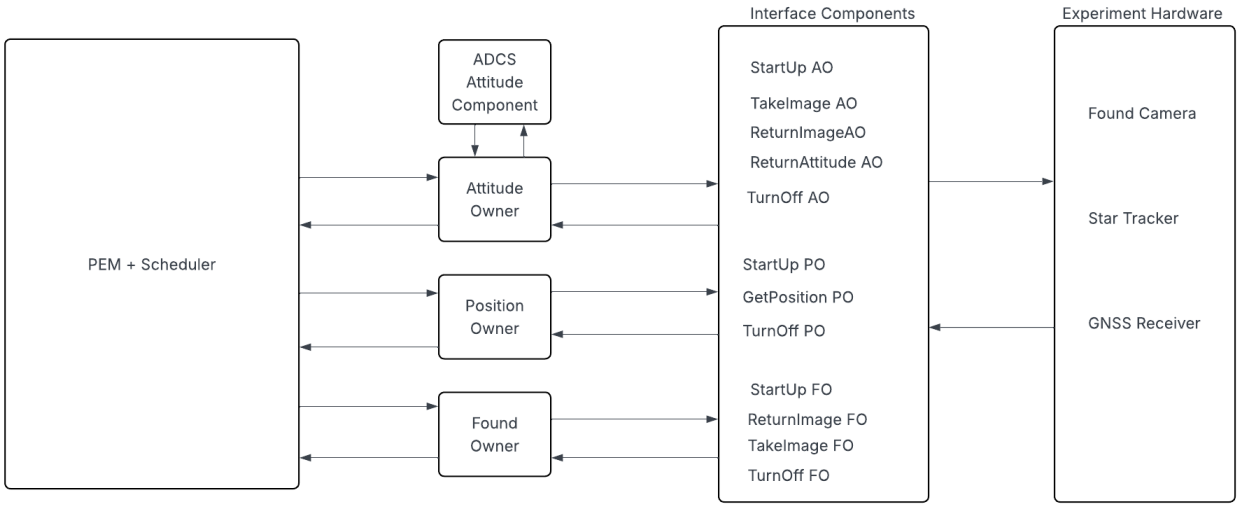

Fig. 3.2: Utility Interface

3.4.1 Attitude Owner

We still have a big attitude owner that is talking to the little bits. This will read the experiment and the frequency to determine how to call the components

Ports

The ports between the Image Scheduler <–> Interface Components <–> Experiment Hardware along with how ADCS will talk to the interface components and the Experiment Hardware

Inputs

The Attitude owner just takes in one part of an experiment as well as a potential ADCS request. It will then create a new struct at the time of receiving either struct or both to make one request.

```
1     struct ExperimentStar {
2         int takeImages;
3         float rate;
4     }
5
```

```
    struct ADCSRequest {
        bool getAttitude
    }
```

Outputs

```
    struct ExperimentOutput {
        arr[n][m] attitude;
    }

    struct experimentOut {
        image* StarImage;
    }

    struct Active {
    bool cameraOn;
    bool experimentActive;
    }
```

Preconditions, Post Conditions, Errors

Preconditions: The Star Tracker is on, no experiment is currently running, check to see if ADCS needs attitude values from the star tracker

Post Conditions: If there are no incoming experiments or requests shut down the camera and this component

Errors: Attitude could not be returned = -1

Camera could not be turned on = -2

Camera could not be turned off = -3

Image could not be taken = -4

Implementation

To make sure the entire topology isn't bogged down the components are active to give them all their own thread to work on

Type 1: The utility is turned off between measurements Type 2: The utility is kept on between measurements Attitude Owner Each owner has a frequency of measurements that decide the state of the utilities between measurements

```
    //Each type contains a way to turn on and
    active component AttitudeOwner {
        void ExperimentIn_handler(port num, Context) {
        if(Camera on && No active experiment) {
            if (Experiment.rate > frequency) {
                enable type 1//Takes images and turns off startracker
                   between images
            }
            else {
                enable type 2//Takes images without turning off star tracker
            }
        }
        }

    }

    Startup() {
        if(startracker off)
        send ON to startracker
    }


//Will command the camera software component to capture space
    takeImage() {
        if (Camera is on) {
            if(No active experiment) {
            Send the command to take the image
            }
        }
        else {
        LogEvent(-4);
        }
    }




    ReturnImage(LOCAL) {
        get Image from star tracker
            passes the image to the LOCAL specified from function call
        }


//The camera software will invoke a input port on the ReturnAttitudeAO
    GetAttitude() {
        take in attitude
        check to see if attitude is valid
            if(valid) {
```

```
                send attitude to ADCS unit
                }
            else {
            LogEvent(-1)
            }


    }

//Done by image scheduler when no experiments are to be run
    turnOffStarTracker() {}
    turn off the camera //some input port?
    if(false) {
    LogEvent(-3)
    }
```

Testing

3.4.2 Position Owner

Inputs

Requests originating from the PEM (Maybe some other component) which constantly requests the GNSS receiver to send a position NOTE: Do we need a request at all? Might it be faster to have some sort of "Universal" component which talks to anything that is runnning constantly. If the GNSS is running constantly, at the whims of the rest of the satellite, we should have global rate group component

```

struct getPosition {
    bool takePosition;

}
```

Outputs

The Position that the GNSS receiver gets

```

struct Position {
    float arr[n][m] Position; //Matrix containing satellite position

}
```

Pre conditions, Post Conditions, Errors

PreConditions: Some device needs a position, there is power available, Receiver is On, Receiver is available to request

PostConditions: Not Enough Power sent to the GNSS Receiver

Errors: GNSS Receiver is not ON = =1 GNSS Receiver returns an error for internal errors = -2

Implementation

```
    active component StartUpPO {

    StartUp_handler(port num, context) {
        if(GNSS-Receiver off)
        send ON to GNSS
    }

}

//The camera software will invoke a input port on the ReturnAttitudeAO
active component ReturnAPositionPO {
    PositionIn_handler(port num, context) {
        take in position
        check to see if position is valid //the matrix is formatted
           correctly?, unsure of                                    //this
            check
            if(valid) {
                send attitude to ADCS unit
                }
            else {
            LogEvent(-1)
            }


    }
}


//Done by image scheduler when no experiments are to be run
active component TurnOffPO {
    turn off the camera //some input port?
    if(Unable) {
    LogEvent(-3)
    }

}
```

3.4.3 Found Owner

Consists of a larger Found Owner which speaks to the smaller sub-components in the sequence in which the

Ports

Inputs

Outputs

3.4.4 Implementation

Testing

TBD

3.5 Science Software

3.6 Payload Storage

3.6.1 LOST

This is an algorithm that utilizes the images taken by the star tracker by using the position of the stars in the image to figure out the attitude of the satellite.

3.6.2 FOUND

This is an algorithm that utilizes the images taken by the Found Camera by using the horizon of the earth in the image to figure out the position of the satellite

Inputs

Takes inputs from LOST and FOUND.

- LOST: Returns the quaternion (this is just a fancy way to say complex number matrix with directions/rotation). Once we get the altitude we will store that in bits in the payload storage. Due to high radiation bit flipping is common and we will have to use error-correcting codes to solve this.

Outputs

TBD

Preconditions, Post Conditions, Errors

TBD

Implementation

TBD

Testing

TBD

3.7 Commercial Software

3.7.1 FOUND Camera Software

3.7.2 Star Tracker Software

3.7.3 GNSS Software

4 Communications

4.1 Hardware

The on-board radio for HuskySat-2 is an Endurosat S-band Transceiver. It is a low power consumption, high output RF power, high sensitivity, software reconfigurable radio transceiver (EnduroSat S-band Transceiver Datasheet). It operates using the EnduroSat Protocol Stack (ESPS) and does the encryption and decryption for HS-2, using the AES256 encryption protocol. The communications hardware establishes connection with the ground station, sends telemetry, and receives commands.

4.2 Database

The communications subsystem stores all commands received in its onboard database [TBD].

4.3 Software

TBD

4.4 Inputs and outputs

4.4.1 Inputs

- Commands (from Ground)
- Health telemetry
- Payload Telemetry

4.4.2 Outputs

- Telemetry for downlink (see Command and Telemetry List)

5 ADCS

There are 4 different types of components in the ADCS software block diagram: hardware, storage, ADCS software, and flight computer software. Hardware is all ADCS hardware, as in all hardware that ADCS directly interfaces with. Storage is maintained by the flight computer and is either read or written by ADCS. ADCS software is all software written by the ADCS team. Flight computer software is written by the flight computer team.

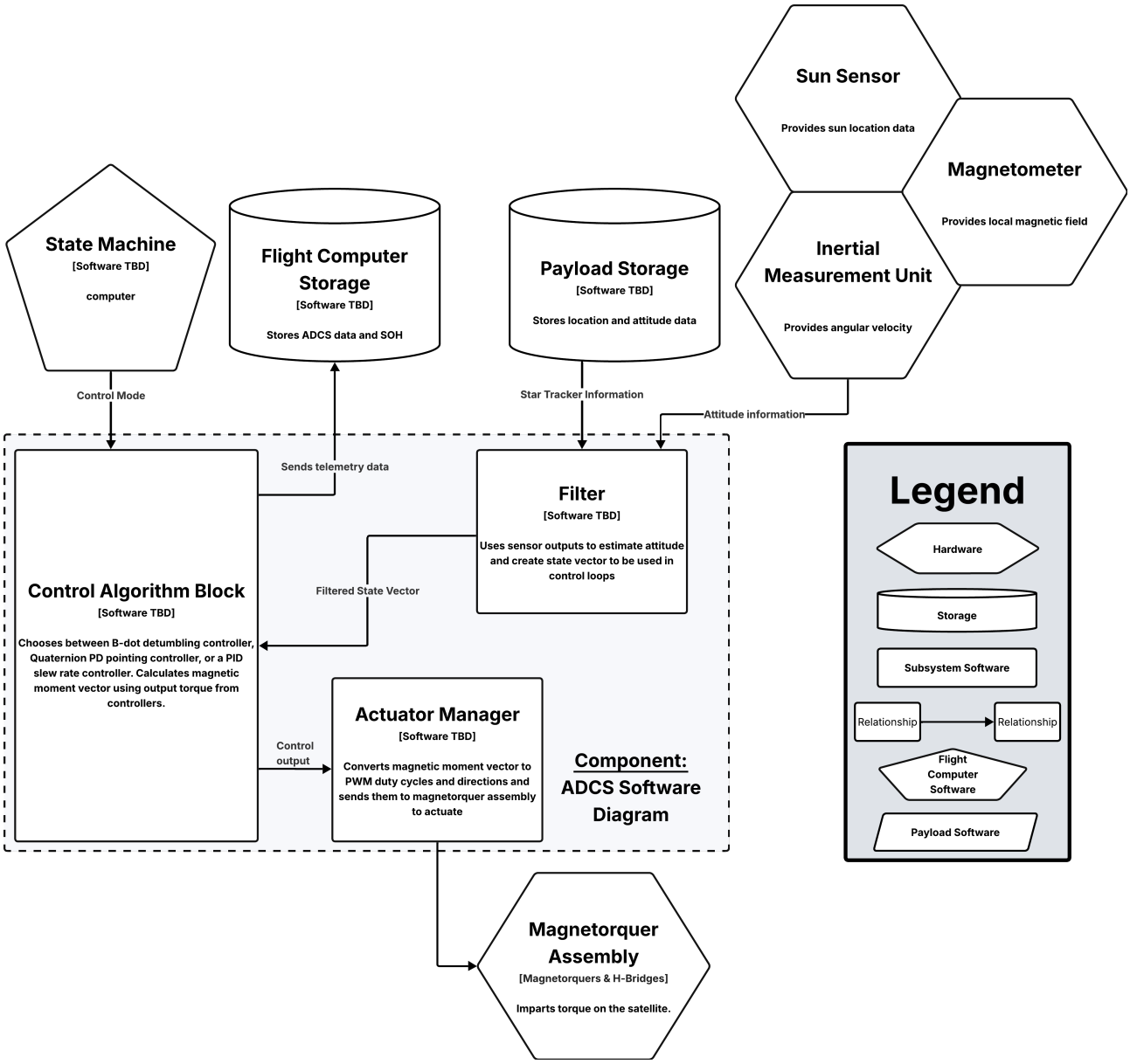

Fig. 5.1: The ADCS software: component level

Definition of Math Variables

Before we begin looking at the different processes inside of ADCS, we will define some of the variables we will be using. These are variables that are going to be defined explicitly and used repeatedly to mean the same thing every time.

Variable	Meaning	Other Names
Subscript of $\mathbf{b/i}$	As measured in the body frame, relative to the inertial frame.	Will be omitted for conciseness.
$\mathbf{a_{b/i}}$	Linear acceleration in the body frame; vector in $\mathbb{R}^3$.	$\mathbf{a}$
$\boldsymbol{\omega}_{\mathbf{b/i}}$	Angular velocity in the body frame; vector in $\mathbb{R}^3$	$\boldsymbol{\omega}$
$\mathbf{B}$	Local magnetic field through the body frame; vector in $\mathbb{R}^3$	—
$\boldsymbol{\beta}$	Earth's magnetic field; vector in $\mathbb{R}^3$.	—
$\mathbf{u}$	Control output; vector.	—
$\mathbf{x}$	Satellite state vector.	—
$\mathbf{m}$	Magnetic dipole from magnetorquers; vector in $\mathbb{R}^3$.	—

Tab. 5.1: ADCS Math Variables

5.1 Filter

[TBR] ADCS will utilize a simple state filter to properly provide the control algorithms the necessary information in a state vector to perform attitude maneuvers.
Note: as it currently stands, the ADCS has access to attitude truth measure data from the payload subsystem. In theory, this would allow the ADCS to perform sensor fusion using an extended Kalman filter. It is unlikely that we will need to do so, but in the case that our attitude estimates from the IMU grow too large (in error), there is a structure there to support more advanced filtering and state estimation.

Sensor Outputs

As the filter relies on sensor outputs, we first define our sensor outputs.
The IMU has 6 degrees of freedom (DOF), meaning we get 6 measurements:

1 X-axis linear acceleration

2 Y-axis linear acceleration

3 Z-axis linear acceleration

4 X-axis angular velocity

5 Y-axis angular velocity

6 Z-axis angular velocity

7 *Estimated* attitude

These measurements can be condensed to 2 vectors and a quaternion; the linear acceleration vector as measured in the body frame can be written as $\mathbf{a_{b/i}}$, and the angular velocity vector can be condensed to $\boldsymbol{\omega}_{\mathbf{b/i}}$. The attitude will be in the form of a unit quaternion $\boldsymbol{q} \in \mathbb{H}$ (Note: there will be no standard variable name for any specific quaternions; they will always be defined when they come up).

The magnetometer measures the local magnetic field through the body frame, i.e., the magnetic field through the X-axis, Y-axis, and Z-axis. This is a vector that we will denote as $\mathbf{B}$.

Sun trackers will be used on HS-2 to provide information on whether a particular side of the satellite is facing the sun or not, meaning that every individual sun tracker measurement will be either 0 or 1. We can condense this to simply being a boolean.

The star tracker aboard HS-2 will provide an attitude in the celestial frame, which will be represented by a unit quaternion $\boldsymbol{q} \in \mathbb{H}$. This will only be sent to the filter in the case that ADCS implements sensor fusion into the filter model.

Inputs

The filter model relies on all ADCS sensor inputs, with the star tracker attitude data being optional. The filter would be given, as inputs,

- IMU Accelerometer Data (linear acceleration), $\mathbf{a}$
- IMU Gyroscope Data (angular velocity), $\boldsymbol{\omega}$
- IMU attitude data (quaternion), $\boldsymbol{q}$
- Magnetometer Magnetic Field Data, $\mathbf{B}$
- Sun sensor data, **boolean**

In practice, the ADCS software will rely on an external library that provides vector, matrix, and quaternion functionality. The sensor outputs would then be actualized as:

Listing 5.1: Filter Input Struct

```
// contains all infromation on how an expirement should be run
struct Filter_Input {
    Vector<3> lin_accel;   // output from IMU accelerometers
    Vector<3> omega;       // output from IMU gyroscopes
    Quaternion q_est;      // output from IMU estimates
    Vector<3> b_field;     // output from magnetometer
    bool sun_data;         // output from sun tracker
}
```

Outputs

The filter model will give an output of a state 'vector' estimate, $\mathbf{x}$. This state vector will contain all necessary information for the control loop, including the angular velocity vector, the attitude *estimate*, magnetic field through the body frame, and the sun sensor data:

Listing 5.2: Filter Output Struct

```
// contains all infromation on how an expirement should be run
struct State_Est {
    Vector<3> omega;       // output from IMU gyroscopes
    Vector<3> b_field;     // output from magnetometer
    Quaternion q_est;      // output from star tracker
    bool sun_data;         // output from sun tracker
}
```

Preconditions, Post Conditions, Warnings

Preconditions must satisfy:

- There must be valid sensor inputs at the expected time sync tolerances.
- System time is initialized.
- Input data must conform to expected measurement format.
- Input data must be numerically stable.

The post conditions must satisfy:

- Output state vector must be numerically stable.

- Output state vector must be properly timestamped.
- Output state vector must be properly sent to control algorithm module.

Warnings:

- Prolonged sensor delays can degrade control performance.
- Non-normalized quantities will degrade control performance.
- Significant differences between sensor clocks and system time can degrade performance and output.
- Re-initializing sensors during runtime can cause issues with filter state output.

Implementation

[TBR] The filter will handle all communications with sensors and with the control algorithm module. It will take in data from sensors, put it into a usable state vector and send it to the control algorithms to then start performing attitude maneuvers. No state estimation will be implemented as it currently stands, and the filter module is more of an information hub for the control algorithms to get information. There is no internal math involved.

Testing

[TBR] The filter will be validated with unit tests, software in the loop testing, and hardware in the loop testing with actual communication with sensors.

5.2 Control Algorithm Block

The control algorithm block will contain the 'brains' of the ADCS software. It will perform all calculations to facilitate the actuation of the satellite. It interfaces with the filter module, the flight computer state machine, and the actuator manager. The control block contains 3 controllers, a b-dot detumbling controller, a quaternion PD pointing controller, a PID angular velocity controller, and a magnetic moment manager.

Inputs

The control block takes in the state vector $\mathbf{x}$ from the filter and parses into the different values it can use for control maneuvers. The flight computer state machine sends in a control mode (detumble, point, or slew rate).

Outputs

The control algorithm block outputs the magnetic moment vector $\mathbf{m}$, which is received by the actuator manager, which converts it into usable PWM signals.

Preconditions, Post Conditions, Warnings

Preconditions:

- There must be valid state vector input.
- There must be a valid control mode input.
- Input data must conform to expected measurement format.
- Input data must be numerically stable.

Post conditions:

- Output magnetic moment vector must be numerically stable.
- Output magnetic moment vector must satisfy physical limitations of magnetorquers.

Warnings:

- Incorrect control gains can cause performance degradation.

Implementation

The implementation section will be split into 4 sections going over the math and implementation behind the 3 controllers and magnetic moment manager aboard HS-2.

Detumbling Controller

The detumbling controller on HS-2 relies on the B-dot control law. It is a feedback controller that brings your angular velocity down to zero. A quick look into the math will shown, as well as other possible implementations of the control law.

For a rotating rigid body through a vector field, the *rate of change* of that vector field through the body frame is proportional to the rotation rate of that rigid body, i.e. $\dot{\mathbf{B}} \sim \boldsymbol{\omega}$. This means we can design a controller that relies only on consecutive measurements of the magnetic field through the body frame:

$$\mathbf{u} = -k\dot{\mathbf{B}}$$

where the dot notation is the time derivative. There is a mathematically equivalent version of the B-dot control law that relies on minimizing your angular velocity through a more analytical solution:

$$\mathbf{u} = k(\boldsymbol{\omega} \times \mathbf{B})$$

Simulations showed no significant differences in the implementations, so, in order to reduce the number of floating-point operations per second (FLOPS), we went with the simpler control law.

Since we went with the simpler control law, code implementation is simplified. We keep track of a previous magnetic field measurement and calculate B-dot using a simple derivative estimate.

Listing 5.3: BDot pseudocode

```
1    Vector<3> compute(Vector<3> current, double dt){
2        Vector<3> b_dot = (current - prev_b_field) / dt
3        prev_b_field = current
4        return (-k * b_dot)
5    }
```

Pointing Controller

The pointing controller is a quaternion PD (proportional-derivative) controller. It is a feedback control law that brings your attitude error (which will be defined soon) down to zero (or more accurately, the notion of 'zero' on $\mathbf{SO}(3)$). First, let us look at some of the math governing our control.

Let the current attitude of the satellite be a unit quaternion $\boldsymbol{q} \in \mathbb{H}$ and our target attitude be (another unit quaternion) $\boldsymbol{p} \in \mathbb{H}$. We can then relate the two quantities using the notion of quaternion rotation:

$$\delta\boldsymbol{q} \cdot \boldsymbol{q} = \boldsymbol{p}$$

where $\delta\boldsymbol{q}$. We can then find an expression in terms of the unknown error value:

$$\delta\boldsymbol{q} = \boldsymbol{p} \cdot \boldsymbol{q}^*$$

So our error calculation will perform the aforementioned calculation.

To build the controller, let us quickly revisit the definition of a quaternion:

$$\boldsymbol{q} = \begin{bmatrix} \mathbf{q}_v \\ q_4 \end{bmatrix}$$

The notion of 'zero' on $\mathbf{SO}(3)$ is the identity matrix, or for quaternions, simply $\boldsymbol{q} = [\mathbf{0} \;\; 1]^\top$. Therefore, for a pointing controller, we can define the proportional part of the PD control loop to bring $\delta\mathbf{q}_v \to \mathbf{0}$. We can add a derivative portion to the controller to get our final control output:

$$\mathbf{u} = -k_p\mathbf{q}_v - k_d\boldsymbol{\omega}$$

There's a lot of math behind *why* this actually works, but we don't need to go into that :-).

Since we are using a library for quaternions, vectors, and matrices, our implementation simplifies, as we do not have to worry about the internals of quaternion multiplication!

Listing 5.4: QuaternionPD pseudocode

```
Quaternion calc_error(Quaternion setpoint,
                      Quaternion current)
{
    return setpoint * current.conjugate
}

Vector<3> compute (Quaternion setpoint,
                   Quaternion current,
                   Vector<3> omega)
{
    Quaternion error = calc_error(setpoint, current)
    Vector<3> error__vec = error.vector
    return (-k_p * error_vec - k_d * omega)
}
```

Slew Rate Controller

The slew rate controller on HS-2 is a PID controller that acts on the measured angular velocity vector, $\boldsymbol{\omega}$. It is a standard PID, with the error calculation being,

$$\boldsymbol{\omega}_{\text{measured}} - \boldsymbol{\omega}_{\mathbf{b/i}} = \boldsymbol{\omega}_e$$

The integral and derivative parts of the controller being computed with estimates. The controller effectively looks like:

$$\mathbf{u} = k_p\boldsymbol{\omega}_e + k_i \int \boldsymbol{\omega}_e \mathrm{d}t - k_d\boldsymbol{\alpha}$$

Listing 5.5: OmegaPID pseudocode

```
1     compute(Vector<3> setpoint,
2             Vector<3> current,
3             double dt)
4     {
5         Vector<3> error = setpoint - current
6         Vector<3> alpha = (current - omega_prev) / dt
7         I_error = I_error + (error * dt)
8         omega_prev = current
9         return ((error * kP) + (_I_error * kI) - (alpha * kD)
10    }
```

Moment manager

[TBR]. It might be necessary that HS-2 implements more advanced control, including the possibility of model predictive control (MPC). This is pending more analysis.

The magnetic moment manager takes the control outputs from the controllers and turns it into an actual magnetic moment value that can be turned into PWM signals by the actuator manager. Since magnetorquers are underactuated, we will never be able to actuate the satellite across all 3 DOF. To help mitigate lost torques and power, we only actuate for the torques that do not line up with Earth's magnetic field, $\boldsymbol{\beta}$.

Our controller outputs can effectively be thought of as the torque that needs to be imparted onto the satellite. The desired torque will be denoted by $\boldsymbol{\tau}_d$, and our actual torque that we can impart on the satellite will be denoted by $\boldsymbol{\tau}_\perp$. We can find the actual torque required by seeing what part of the torque does not align with the magnetic field:

$$\boldsymbol{\tau}_\perp = \boldsymbol{\tau}_d - (\boldsymbol{\tau}_d \cdot \hat{\boldsymbol{\beta}})\hat{\boldsymbol{\beta}}$$

where $\hat{\boldsymbol{\beta}}$ is the normalized magnetic field vector. This is done for power conservation, as sending power through the magnetorquers when we can't actually actuate in that direction is wasteful and unnecessary.

We can find the magnetic moment by taking the cross product of the torque we can generate and the magnetic field, i.e.

$$\mathbf{m} = \hat{\boldsymbol{\beta}} \times \boldsymbol{\tau}_\perp$$

The moment manager also ensures that moments fit into the specs of the magnetorquers by clamping values that are too large.

Testing

The control algorithms will be tested with unit tests and software in the loop simulations.

5.3 Actuator Manager

The ADCS actuator manager takes in required magnetic moments, turns them to PWMs, and sends them to the magnetorquers. It interfaces with the flight computer PWM pins as well the direction pins (general purpose IO pins that can be written HIGH or LOW).

Inputs

The actuator manager takes in the the magnetic moment vector, $\mathbf{m}$, from the controller.

Outputs

The outputs are direct data through the PWM and GPIO pins of the flight computer. The PWM pins basically output a duty cycle, and the GPIO pins are the pins that dictate the direction of current flow through the magnetorquers; they can be written as either HIGH or LOW.

Preconditions, Post Conditions, Warnings

Preconditions:

- There must be valid moment vector input.

Post conditions:

- Output PWMs must be stable.

Warnings:

- Incorrect moment$\leftrightarrow$PWM conversion can cause performance degradation.

Implementation

The actuator manager will need direct connections to the pins on the flight computer. The magnetorquers currently chosen for HS-2 will have a 0.2Am^2 nominal magnetic moment. This corresponds to a PWM duty cycle of 100%. An input moment on 0 corresponds to a PWM duty cycle of 0%. The sign of

the input informs what direction the current should flow in, and therefore, whether the direction pin should be written HIGH or LOW.

Testing

The actuator manager will undergo hardware in the loop testing.

6 Electrical Power System (EPS)

Electrical Power System (EPS) related software on HS-2 is dedicated to fault management and ensuring the battery is able to charge in an optimal manner.

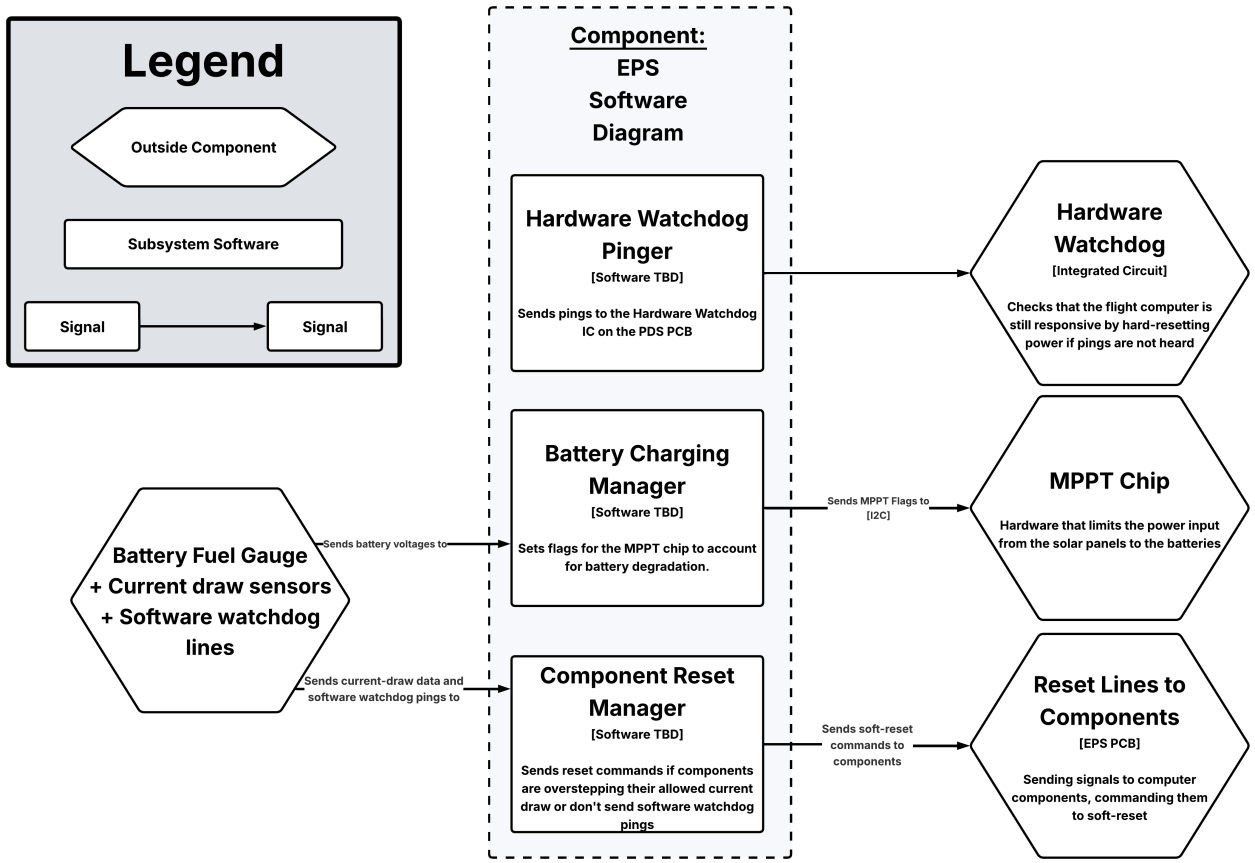

Fig. 6.1: The EPS software: component level

6.1 Component Reset Manager

[TBR] The Component Reset Manager sends reset signals to integrated circuits if it determines that they are drawing unsafe amounts of current. Additionally, if the EPS & CDH are subsystems determine that additional software-based watchdog timers are necessary (in addition to a hardware watchdog timer IC), the Component Reset Manager will also encompass software watchdog functionality.

Testing

TBD

6.2 Battery Charging Manager

[TBR] The buck-boost converter on the Battery Management System PCB is responsible for Maximum Power Point Tracking. This MPPT chip has several settable flags, (mainly voltage thresholds for various charging modes), which need to be updated as the satellite's batteries degrade due to lithium plating.

6.3 Hardware Watchdog Pinger

[TBR] To prevent the hardware watchdog from resetting the flight computer, an application on the flight computer needs to periodically send signals to the hardware watchdog. In the event that the flight computer loses functionality, the hardware watchdog will cycle power to the flight computer.

7 Thermal

There are 4 different types of components within the thermal subsystem: hardware, thermal software, storage, and flight computer software. Hardware includes all thermal control hardware, such as Kapton heaters, temperature sensors, and heater amplifiers, directly interfaced by the thermal system. Storage is managed by the flight computer and is either read from or written to by the thermal software for logging temperature data and control parameters. Thermal software encompasses all code developed by the thermal team to regulate satellite temperature. Flight computer software, written by the CDH team, provides the runtime environment and interfaces with other subsystems like EPS.

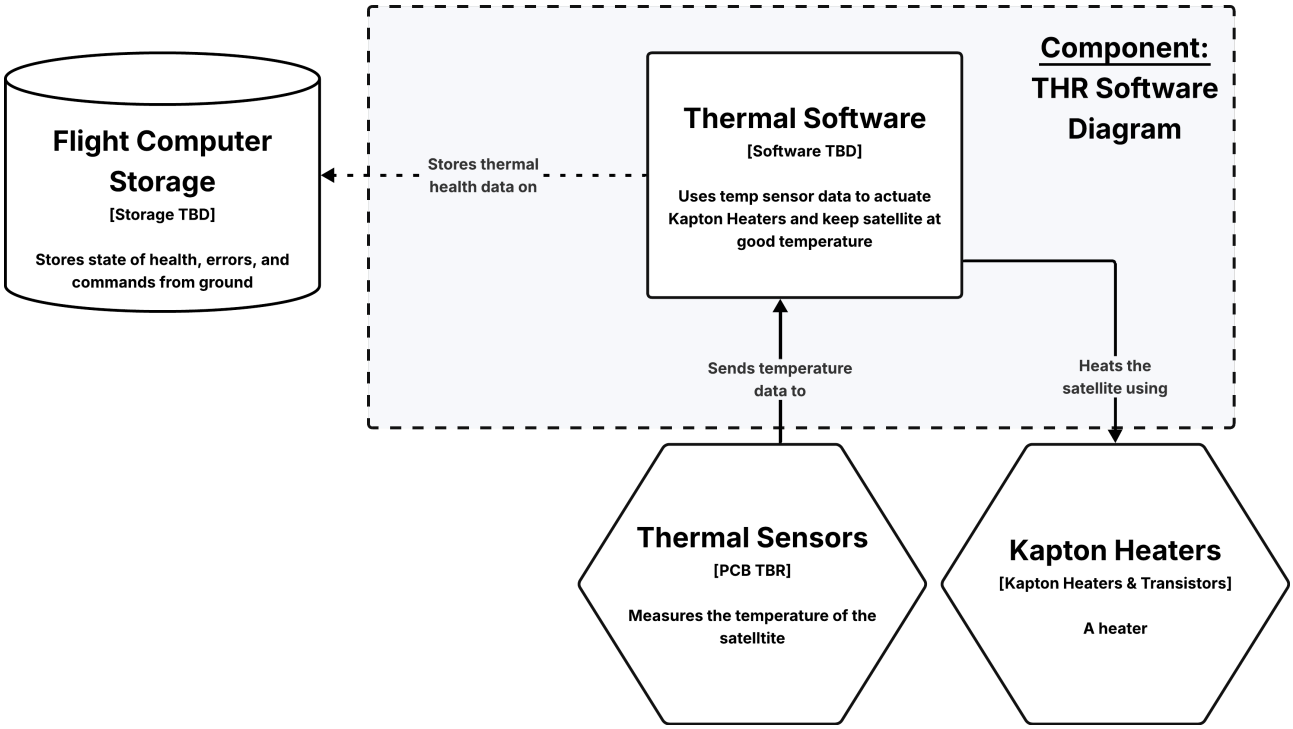

Fig. 7.1: The Thermal software: component level

Definition of Math Variables

Before we begin looking at the different processes inside of THR, we will define some of the variables we will be using. These are variables that are going to be defined explicitly and used repeatedly to ensure continuity in thermal modeling and control softwares.

Variable	Meaning	Other Names
Subscript of $\mathbf{T_b}$	Temperature of the satellite body measured in degrees Celsius	T_{body}
T_{set}	Target temperature setpoint for thermal control, in C.	–
ΔT	Temperature difference between current and setpoint	ω
$\P_{heat}$	Power supplied to heaters, in watts (W)	—

Tab. 7.1: Thermal Math Variables

7.1 Thermal Control Algorithm

[TBR] The Thermal control algorithm employs a PID controller to regulate the EPS battery pack temperature, utilizing data from temperature sensors to activate them.

Sensor Outputs

The thermal control relies on sensor outputs for temperature regulation.

Temperature sensors, distributed across the satellite, provide analog voltage readings convertible to temperature in Celsius. The primary sensor locations include:

1 Battery pack temperature sensor

2 Solar panel temperature sensors

3 Internal component temperature sensors

4 External surface temperature sensors

These measurements provide the thermal control system with real-time temperature data for each monitored zone.

Inputs

The thermal control algorithm relies on temperature sensor inputs and system state information. The control system receives:

- Temperature sensor data (multiple zones), T_{zones}
- Temperature setpoints, T_{set}

- Heater status feedback, **boolean**
- Power system status from EPS

Outputs

TBD

Preconditions, Post Conditions, Warnings

TBD

Implementation

TBD

Testing

TBD

8 Kernel

8.1 Threads

Priorities of the software threads in the RTOS range from 10 (highest) to 0 (lowest), and rates are set to balance responsiveness with power constraints. Currently, the CD&H team needs to do more investigation to understand how OS-level scheduling interacts with message queuing/scheduling in F Prime (or another framework such as NASA cFS).

Thread Name	**Priority**	**Priority Number**	**Rate (Hz)**
ADCS Filter	TBD	TBD	TBD
ADCS Control Algorithm Block	TBD	TBD	TBD
ADCS Actuator Manager	TBD	TBD	TBD
Payload Experiment Manager	TBD	TBD	TBD
Payload Image Scheduler	TBD	TBD	TBD
Payload Image Processor	TBD	TBD	TBD
Payload Attitude Owner	TBD	TBD	TBD
Payload Position Owner	TBD	TBD	TBD
Thermal Control Algorithm	TBD	TBD	TBD

Tab. 8.1: System Threads Overview

List of Figures

List of Tables