PAN Software Systems Manual

PAN Software Team

Oct 21, 2021

CONTENTS:

This manual contains documentation for software written for the Pathfinder for Autonomous Navigation (PAN) project, a part of Space System Design Studio.

CHAPTER

ONE

FLIGHT SOFTWARE AND SYSTEMS DOCUMENTATION

This section of the documentation contains documentation on both the flight software and the PAN satellite's subsystem architectures. The two ideas go hand-in-hand, which is why their documentation is woven together. This section is primarily meant to be *design* documentation, although there is some user documentation as well.

1.1 Installing Flight Software

The code for flight software is available here. Follow the instructions in the README under `ptest` to set up the code for development and testing.

FlightSoftware is dependent on a build system called PlatformIO. See *Flight Software Build System* to see how this system is set up for our use.

1.2 Structure of Flight Software Repository and Documentation

This repository is very monolithic and actually contains three separate products:

- FlightSoftware, which runs on the Flight Controller of our spacecraft
- ADCSSoftware, which runs on the Attitude Control subsystem of our spacecraft
- PTest, a hardware-out-of-the-loop (HOOTL) and hardware-in-the-loop (HITL) testing platform designed to provide mission-fidelity testing.

Unless otherwise specified, this documentation talks about the design of Flight Software, not the other platforms. Documentation for ADCSSoftware may be forthcoming, but for now it suffices to know that it exists to service the hardware described in *Attitude Determination and Control*. However, because PTest is an extensive platform that is still being heavily developed, we are producing good documentation for it here (see the table of contents below.)

1.2.1 Flight Software Components

Control Tasks

Type name: `Serializer<bool>`

The core unit of work in PAN flight software is the *control task*. The control task interface specifies one function, `execute`, that can have any return type but accepts no arguments.

How does `execute` know what data to act upon? Upon construction, the control task is responsible for either creating or finding *state fields*. Created state fields correspond to the outputs of the control task, and found state fields correspond to the inputs.

For example, the `ClockManager` control task creates a state field housing the current time, and updates this value on every execution of the control cycle. A control task "finds" a state field by querying the State Field Registry; see more about state fields and the state field registry below.

State Fields

A state field is nothing more than a wrapper around a simple type, along with a `get` and `set` function. All data representing important values on the spacecraft are stored in state fields, which are centrally indexed by the state field registry (see below). The indexing is by string; all state fields have a unique name, accessible by the function `name`.

There are three kinds of state fields on the spacecraft:

- **Readable** state fields are fields whose values can be read from the ground but whose values cannot be modified from the ground.
- **Writable** state fields are fields whose values can both be read from and modified from the ground.
- **Internal** state fields are implementation details. This kind of state field exists so that control tasks can share data across each other without breaking the encapsulated design of a control task. We care about encapsulation because it enables unit testability.

Internal state fields can have any underlying data type, but since readable and writable state fields need to be passable over the limited-bandwidth radio that we have onboard the spacecraft, their types are restricted to the following list:

- `bool`, `unsigned int`, `unsigned char`, `signed int`, `signed char`, `float`, `double`
- `gps_time_t`: A GPS time class, which has the sub-members `wn` (week number), `tow` (time of week in milliseconds, and `ns`, which is an offset of +/- 1000000 nanoseconds off of the time of week.
- `f_vector_t`, `d_vector_t`: Vectors, which are nothing more than a renaming of `std::array<float, 3>` or `std::array<double, 3>`.
- `lin::Vector3f`, `lin::Vector3d`: These are custom-built vector classes that facilitate easy computation, much of it at compile-time. See this for more information on these utilities.
- `f_quat_t`, `d_quat_t`: Quaternions, which are nothing more than a renaming of `std::array<float, 4>` or `std::array<double, 4>`.
- `lin::Vector4f`, `lin::Vector4d`: Custom-built quaternion classes, again provided by `lin`.

In order to encode and decode state fields from a string representation so that they can be transmitted over the radio, readable and writable state fields contain a `Serializer` object. The control tasks that manage telemetry use the functions contained within the serializer to manage the value of a readable/writable state field. Check out more info about *Serializers*.

Faults

TODO: COMPLETE DOCUMENTATION

Faults serve are a way to modifiy the behavior of the satellite in a fundamental, off-nomial way. Faults are declared and signaled / unsignaled in the ControlTask that is most closely tied to the data that can determine a fault condition. For **PAN**, since we are choosing to only use Faults on hardware failure, this means Faults are declared in the device monitor Control Tasks.

Upon construction, Faults must be tied to a name (so that it can be located in the SFR), a *persistence*, a number of consecutive signals that are required after which the next signal will trip the fault. Faults must also be provided a control_cycle_count, which to prevent multiple signals on the same cycle.

Faults can be signaled through the member function `signal()`, which increments a private internal counter *num_consecutive_signals*. `signal()` should be called whenever the Fault condition is true. If the fault condition is not met during a control cycle, the `unsignal()` function should be called to reset `num_consecutive_signals` to 0.

Faults themselves encapsulate N different fields that are implemented as statefield

- A boolean statefield that represents whether or not it is *faulted*
- Persistence

The State Field Registry

The state field registry contains lists of pointers to events, faults, and readable, writable, and internal state fields, along with functions to find and add events', faults', state fields' pointers to the registry.

The purpose of the registry is to enable encapsulation. Upon construction, control tasks receive a reference to a State Field Registry object that is shared across all control tasks. The control tasks can then publish their outputs to other control tasks by adding state fields to the registry, and can find their inputs (which would've been published by other control tasks) from the registry. This makes unit testing each control task extremely simple.

Flight Software Cycle

In order to maintain determinism and reduce complexity in the behavior of Flight Software, the main event loop of the Flight Software, which we call the *flight software cycle*, is single-threaded and deterministically runs Control Tasks, one after the other. The general structure of this loop is read-compute-actuate, as in most robot control loops. It is implemented in *MainControlLoop.cpp*.

In order to maintain deterministic separation between consecutive executions of one control task, there's an "offset" field, which describes the time at which the control task is expected to start, relative to the start of the control cycle.

This offset is enforced using a busy-wait before each control task that waits for the current time to be greater than the offset. If, for some reason, a control task's execution runs into the offset of the next control task, the next control task will begin immediately. This is never expected to occur, though, since we test the control cycle timing thoroughly.

Software Cycle in Flight Code vs HOOTL/HITL Code

This is the flight version of the software cycle. In the HOOTL/HITL (hardware-out-of-the-loop/hardware-in-the-loop) versions of the cycle, there is an additional `DebugTask` that runs after the ClockManager. Its purpose is to exchange state field data with the simulation. The `DebugTask` is required to last at most 50 milliseconds.

Debug Console

The `DebugTask` makes use of a utility that we call the *debug console*. The debug console manages input/output via the USB serial port located on the Teensy. It has two functions: transacting state field values with a simulation computer, and to serve as a general-purpose logging utility for software. Log messages can be written using an exposed function called `printf`, which behaves in the same way as standard `printf` except for one parameter called the *severity* of the message. The following are the available severity levels (adapted from here):

- `debug`: Information useful to developers for debugging the application.
- `info`: Normal operational messages that require no action.
- `notice`: Events that are unusual, but not error conditions.
- `warning`: May indicate that an error will occur if action is not taken.
- `error`: Error conditions

- `critical`: Critical error conditions
- `alert`: Should be corrected immediately
- `emergency`: System is unusable.

1.2.2 Desktop Operation of Flight Software

See *PTest Software Stack* before reading this document.

One of the great things about PAN's flight software is that it can run both on a tiny ARM-based microcontroller (the Teensy 3.5/3.6) and on any Linux or Mac platform. We call the latter version of flight software the *native binary*, because our build system, PlatformIO, calls desktop platforms the "native" platform.

Here we describe some of the pros and cons of using desktop vs. flight software and the implementation differences between the two versions of flight software.

Pros and Cons

Pros:

- Being able to run the spacecraft entirely on your computer enables faster development iteration time.
- Simulation software cannot tell a difference between a connected Teensy and a desktop executable version of the flight software, so this enables a greater guarantee that if the flight software is running correctly on the desktop, then it will also run fine on a Teensy.

Cons:

- Native execution of Flight Software can only happen on Linux or Mac because PTY connections, which are needed in order to connect to the Flight Software binary, are not supported on Windows. This is not a huge development hurdle, given that most of the PAN Flight Software team works on one of these two systems.
- Flight Software runs 2 threads at 100% CPU utilization (see why below), so a minimum of 5 dedicated processor hyperthreads are required for running a full mission simulation:
  - One for the Python-MATLAB simulation interface
  - Two for each flight software binary. If running a leader and follower spacecraft, this amounts to 4 total threads for flight software.

  Additional processing power is still required for the following tasks, which individually consume low CPU bandwidth but require power on the order of ~1 hyperthread:
  - Background MATLAB execution
  - The downlink consumer helper process
  - The uplink producer helper process
  - The user-facing state command prompt
  - Datastores and loggers for simulation data

These requirements can be reduced by running the simulation in single-satellite mode, disabling the MATLAB simulation, or switching out one of the flight software simulations for a Teensy. However, the above is still doable on any hyperthreaded quad-core system, like most recent 15" versions of the MacBook Pro.

Implementation Differences

Clock Management

Microcontrollers have the benefit of a built-in system clock that ticks once per clock cycle, and thus ticks at a deterministic, relatively reliable rate. Modern computers change their clock speeds all the time, so ensuring real-time control for processes on these computers is either difficult or impossible.

On the Teensy, timing is provided via the Arduino functions `micros`, `millis`, and `delay`. We achieve the same with `std::chrono` on desktop compilations of flight software! In order to ensure real-time constraints, rather than using `this_thread::sleep` for pausing the event loop execution, we simply busy-wait until the desired time. This allows for relatively good control cycle timing, at the expense of requiring the main event loop to run with 100% CPU utilization. This is not much of an expense at all, unless the system on which the flight software is running happens to be underpowered.

Communicating with Flight Software

One key difference in how the desktop and Teensy flight software versions differ is how they manage communication with the simulation server in a testing configuration.

When the simulation computer communicates with Flight Software on a Teensy in a TITL/HITL/VITL configuration, it does so over a USB connection. In the HOOTL configuration, the USB connection is replaced by `stdin`/`stdout`, a PTY session to the running flight software process, and a Python-based serial connection to this PTY session.

This requires adding an additional thread to flight software that continuously manages `stdin`/`stdout`, to mimic how the microcontroller has an independent, parallel-executing serial controller that dumps incoming and outgoing data into buffers. This thread runs at 100% CPU utilization because it is constantly buffering the `stdin`/`stdout` in order to meet real-time constraints.

Hardware Management

Any I/O communications with flight hardware have been turned into no-ops, so by default hardware always returns dummy values as the result of any device interactions. The interactions are mocked just enough so that the flight software does not go into a fault state because hardware is missing.

1.2.3 Serializers

Serializers are the workhorse unit of telemetry processing. Each state field that's readable or writable from the ground houses an instance of a serializer, which can be used to convert the value of the state field to/from a bitstream representation.

You read that right: I said *bitstream* representation. Due to the limited 70-byte downlink packet size, PAN serializers take compression to an extreme degree and squeeze every available bit out of the bandwidth. This is achieved by realizing that fields on the spacecraft have a limited range of values, or have a limited range of resolution that we care about on the ground. Using this realization, we can specify a fixed-point scheme for compressing state fields at compile time.

As an example, suppose we have an unsigned integer `x` representing the state of a state machine that has 11 total states. Since 11 is less than 16, note that we only need four bits to actually represent the value of the state field. So a serializer for `x` would compress x down to 4 bits, rather than the 32 bits that it would usually take, or the 8 bits that it would take if we naively reduced x down to an unsigned character. Using schemes like this all across our spacecraft, we've found that compressing at the bit level reduces our telemetry size by up to 60%.

Serializers allow conversion to/from a bitstream representation, but also allow conversions to/from an ASCII representation of the internally containe data. This is useful for transacting data over a USB link when operating the flight software in testing mode. In summary, the following methods are exposed by Serializer:

- `serialize`: Converts a given value into a bitstream, which is stored internally within the Serializer object.
- `deserialize`: Converts the internally stored bitstream into a value and writes it to an input pointer.
- `deserialize`: An overload of deserialize takes the ASCII-encoded value provided in an input character buffer and converts the given value into a bitstream, which is then internally stored.
- `get_bit_array`: Gets a reference to the internal bitstream (this is useful for downlinking).
- `set_bit_array`: Sets the internal bitstream (useful for uplinks.)
- `print`: Converts the internally stored bitstream into an ASCII value that can be printed to a screen.

Constructing a serializer requires specifying the number of bits desired in the representation of its value, along with "minimum" and "maximum" parameters specifying the bounds of the value. For certain serializers (this will be explained in the upcoming sections), there are available default parameters that preclude the need for specifying some of these three values.

Serializer is defined for the following basic types, which are explained in more detail in the hyperlinked sections:

- *Boolean Serializer*
- *Integer Serializer*
- *GPS Time Serializer*
- *Quaternion Serializer*
- *Vector Serializer*

Boolean Serializer

Type name: `Serializer<bool>`

Booleans are the simplest serializer to implement: a boolean's value is either a 1 or a 0, so it can be represented by a bitstream of size 1. The constructor for a boolean serializer accepts no arguments since none are required.

Integer Serializer

Type names: `Serializer<unsigned int>`, `Serializer<signed int>`, `Serializer<unsigned char>`, `Serializer<signed char>`

I described integer serializers in some detail in the introductory section, but here I'll go into greater detail.

There are a few kinds of constructors for integer serializers:

- There's the "standard" constructor that requires three arguments, `min`, `max`, and `bitsize`.
- There's a constructor accepting only `min` and `max`, which automatically computes the required bitsize needed to represent the full range of the possible integer values (i.e. `bitsize` = $\lceil \log_2(\texttt{max} - \texttt{min}) \rceil$).
- For *unsigned int* and *unsigned char* serializers, there's a constructor accepting only `max`. Internally this just calls the constructor we just mentioned, but sets `min` to 0.
- Also for *unsigned int* and *unsigned char* serializers, there's a no-argument constructor that sets `max` to 2^32 - 1 for `unsigned int` serializers and to 2^8 - 1 for `unsigned char` serializers.

These serializers, in general, work as follows: the specified `bitsize` provides

GPS Time Serializer

Type name: `Serializer<gps_time_t>` TODO

Quaternion Serializer

Type name: `Serializer<f_quat_t>`, `Serializer<d_quat_t>` TODO

Vector Serializer

Type name: `Serializer<f_vec_t>`, `Serializer<d_vec_t>` TODO

1.2.4 Flight Software Build System

Flight Software's build system is based on PlatformIO and provides the following set of environments:

- `fsw_native_[leader|follower]` : HOOTL environments. The leader/follower monikers exist because the two environments compile slightly different constants for the hardware-defined leader/follower spacecrafts (not to be confused with the software designation of leader/follower for each spacecraft.) In practice, we almost always use `fsw_native_leader` for HOOTLs as a matter of convention, since the hardware constants do not affect the HOOTL much.
- `fsw_teensy3[5|6]_hitl_[leader|follower]` : HITL environments for both Teensy 3.5 and Teensy 3.6.
- `fsw_native_ci` : Flightless environment that only exists so that symbol-based debugging of unit tests is possible.
- `fsw_flight_[leader|follower]`: Flight code for leader and follower spacecraft.
- `gsw_downlink_parser`: Parses the incoming binary data packets from the spacecraft telemetry into intelligible JSON for consumption by ground software systems.
- `gsw_uplink_producer`: Produces a binary-encoded packet of uplink data based on a user-specified JSON list of state fields and values for those fields.
- `gsw_telem_info_generator`: This purely informational environment produces a utility for listing the telemetry values and associated telemetry flows that exist on the spacecraft. This utility is useful for reviewing telemetry periodically for correctness.

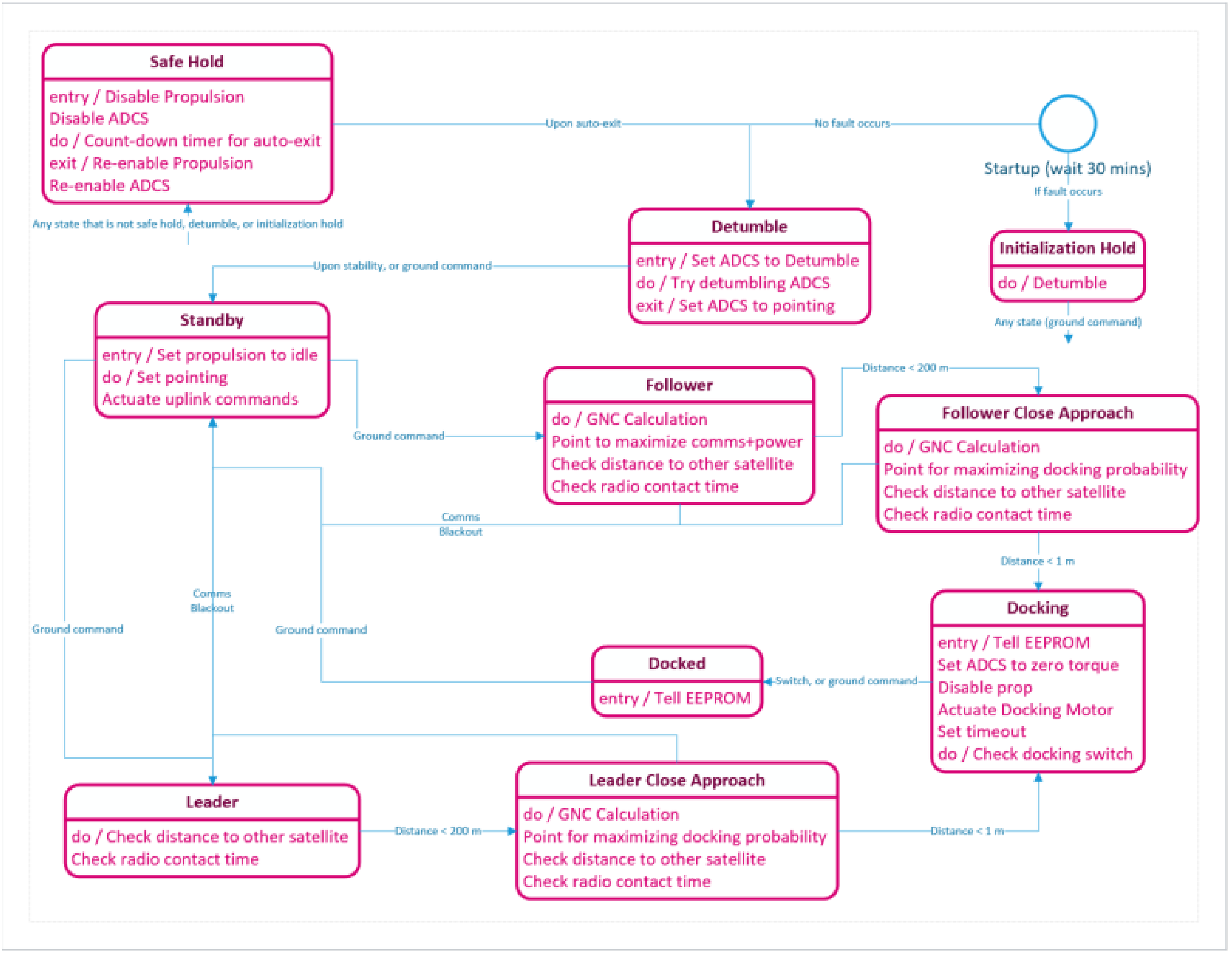
The following compiler macros are used liberally across flight software to conditionally compile certain parts of the codebase in certain environments:

- `DESKTOP` macro, when used, indicates a certain piece of code should only be compiled for `fsw_native*` environments or `gsw*` environments.
- `FUNCTIONAL_TEST` macro, when used, indicates a certain piece of code should only be compiled for HOOTL/HITL, but not flight.
- `FLIGHT` macro, when used, indicates a certain piece of code should only be compiled for flight, but not HOOTL/HITL.
- `GSW` macro, when used, indicates a certain piece of code should only be compiled for `gsw*` environments.

1.2.5 Mission Management

PAN has a very simple goal: dock two satellites together in space. The execution of this goal, though, is made complex by way of fault management and contractual requirements. The *mission manager* software on PAN handles these complexities, by way of the `MissionManager` control task (see _Components). It contains the logic for the PAN Mission State Machine, which controls the overall pointing and orbital maneuver strategy for the spacecraft, and addresses faults, startup, and reboot conditions.

Nominal Mission Management



The PAN Mission Manager contains the following states:

- The "nominal" states, which would be generic states that might be found on any satellite.
  - **Power up**: This is the first state that occurs upon initialization of the first control cycle. During this mode, each subsystem checks the state of its hardware devices, persistent boot values are pulled from EEPROM (see *Persistent State*), and the satellite waits for the end of the contractually-required 30-minute deployment period. It then determines the next mode, which is either "initialization hold" or "detumble".
  - **Initialization hold**: If something is wrong with the spacecraft that would impede its ability to make communications with the ground, this mode tries to get the satellite to stop tumbling and generally conserve power to maximize the probability of communication. Upon successful communication, the ground is free to command the satellite into any mode.

  - **Detumble**: During this mode the satellite is reducing its angular rate so that the attitude of the spacecraft is stabilized.
  - **Standby mode**: No GNC commands are made, and the PAN satellite is put into a power-maximizing orientation by default. A ground command can move the satellite into any state from this state, but the nominal choices are either the "follower" or the "leader" state.

- The "PAN-specific" states, which are very specific to the PAN mission.
  - **Follower**: During this state the satellite points itself to maximize comms + power, and executes propulsion manuevers that match its orbit and phase with the leader and satellite. The leader's position is continuously provided via ground uplink to the follower.
  - **Leader**: This state is the same as the follower state except that propulsion commands are disabled.
  - **Follower Close Approach**: During this state the satellite does the same things as in the "follower" state except it points towards the other satellite.

    Once the two satellites are fairly close together, they both move into the "Docking" state. This determination is dependent on the follower and leader satellites achieving CDGPS lock (see *GPS Receiver*).
  - **Leader Close Approach**: This state is the same as the leader close approach state except that propulsion commands are disabled.
  - **Docking**: The satellites drift towards each other passively (no propulsive or attitude guidance is applied), with the magnetic docking ports on the ends of the satellites causing the satellites to dock.
  - **Docked**: The satellites are joined together; this state is either autonomously entered by the pressing of the docking switch following the "Docking" state, or is a state entered via a ground command (i.e. the ground software notices that the satellites have been in the same position for a long time, and therefore must be docked.)

We can think of the PAN-specific mission states as having two distinct phases: the "active mission" phase, consisting of the follower and leader states, and the "post mission" phase, consisting of the docking and docked states. I'm making this distinction because the behavior of states in these two phases differs if the satellite powers down unexpectedly due to software faults or power faults. Namely, if the satellite powers down during the active mission phase, then the entire mission is restarted from both satellites being in standby, but if power down happens during the post-mission phase, then the satellite restarts from its most recent mission state. This is achieved via saving the satellite state to EEPROM.

To achieve the behaviors specified in each of these states, the mission manager sets states of the spacecraft subsystems (propulsion, radio, ADCS, and docking system) to achieve its desired behavior.

1.2.6 Fault Management

Fault management is a general term for what is actually three distinct functions: fault detection, fault isolation, and fault response. Together, these three behaviors are known in the spacecraft industry as FDIR (fault detection, isolation and response.) Our spacecraft is not very complex–we don't have multiple levels of subsystems–so we generally do not need to isolate root-level faults from their top-level diagnosis. Instead, we only participate in fault detection and response, which I'll describe in greater detail below.

Fault Detection

We detect faults in two main ways: via state machines, and via the Fault class. State machine fault detection is used for complex, tiered fault scenarios where a system is counted as "faulted" due to a history of different kinds of failures; see Quake Fault Handler and Piksi Fault Handler below. But for the most part, we use the Fault class for simpler kinds of failures such as wheel failures or critically low battery levels.

The fault class is just a wrapper around a boolean state field that creates a set of “control” state fields around the boolean value. The boolean value can be signaled or unsignaled by control tasks; if the value is signaled a certain number of times, called the *persistence threshold*, the fault is said to be in a “faulted” state. Ground-controllable parameters can suppress or force this “faulted” state, giving the ground full control over the effect of faults on flight software behavior.

For a fault whose base-level flag is called “x”, it creates the following state fields: TODO describe fault class flags

The following failures are surrounded by a fault class:

- HAVT-reported failures of the ADCS wheel ADCs
- HAVT-reported failure of the ADCS wheel potentiometer
- Critically low battery level
- Prop overpressure
- Prop failed-to-pressurize

For certain kinds of faults, there is not a single fault-detection-and-response event, but several in succession, each depending on the previous detection. The Quake and Piksi fault responses have this property, and so they implement state machines for keeping track of their fault state. I’ll describe these now.

Quake Fault Handler

This handler exists to manage long durations of comms blackouts. The fault response can be described as follows:

- If there has been no comms with the Iridium satellite network for more than 24 hours, force an exit of the mission and Instead go into standby mode in order to be in a comms-maximizing attitude.

  Note that I said “comms with the Iridium satellite network” and *not* “comms with the ground.” The motivation behind this fault handler is that lack of comms might be due to a pointing or a device failure; we should not punish the mission trajectory if its attitude has been unlucky enough that it can establish comms but not send a full downlink packet.
- If the satellite has been in standby due to the previously described fault response for more than 8 hours, trigger a powercycle of the radio. Repeat this fault detection and response up to three times before moving to the next fault response.
- If there continues to be a comms blackout, change the mission state to safehold. The goal with this fault response is to disable ADCS and force the satellite into a random, slow tumble, during which we might hope to get comms. This is useful since it’s possible that a faulty pointing strategy or an erroneous attitude estimation causes a lack of comms in standby mode.

Piksi Fault Handler

This handler manages long durations during which we are unable to establish a real-time kinematic (RTK) GPS fixes. The fault handler recommends moving the satellite to standby if any of the following three situations occur:

- If the Piksi has been unable to collect GPS data for a extended duration (i.e the Piksi is dead)
- If the satellite are in close approach, but the Piksi has been unable to get an RTK fix for a configurable wait period.
- If the satellites are in a close approach state and the Piksi has been able to get RTK fixes while in close approach, but then the Piksi stops getting RTK fixes for a configurable wait period.

Fault Response

The fault response to any fault consists in a change of mission state and a string of subsystem-specific actions. Either of these values may be null; it may be that the response to a fault may recommend no change in mission state or no subsystem-specific actions.

The fault response is administered via a control task called *MainFaultHandler* contained within MissionManager. At a high level, what it does is run a bunch of sub-tasks that each recommend a mission state to *MainFaultHandler* and run subsystem-specific actions. The *MainFaultHandler* combines these recommended mission states into a single recommendation to the MissionManager, in the following priority: - If safehold is recommended by any sub-fault handler, the *MainFaultHandler* recommends safehold - Otherwise, if standby is recommended by any sub-fault handler, the *MainFaultHandler* recommends standby - Otherwise, the *MainFaultHandler* makes no recommendation.

All fault response control tasks derive from a class called *FaultHandlerMachine*, whose inheritance diagram is shown below:

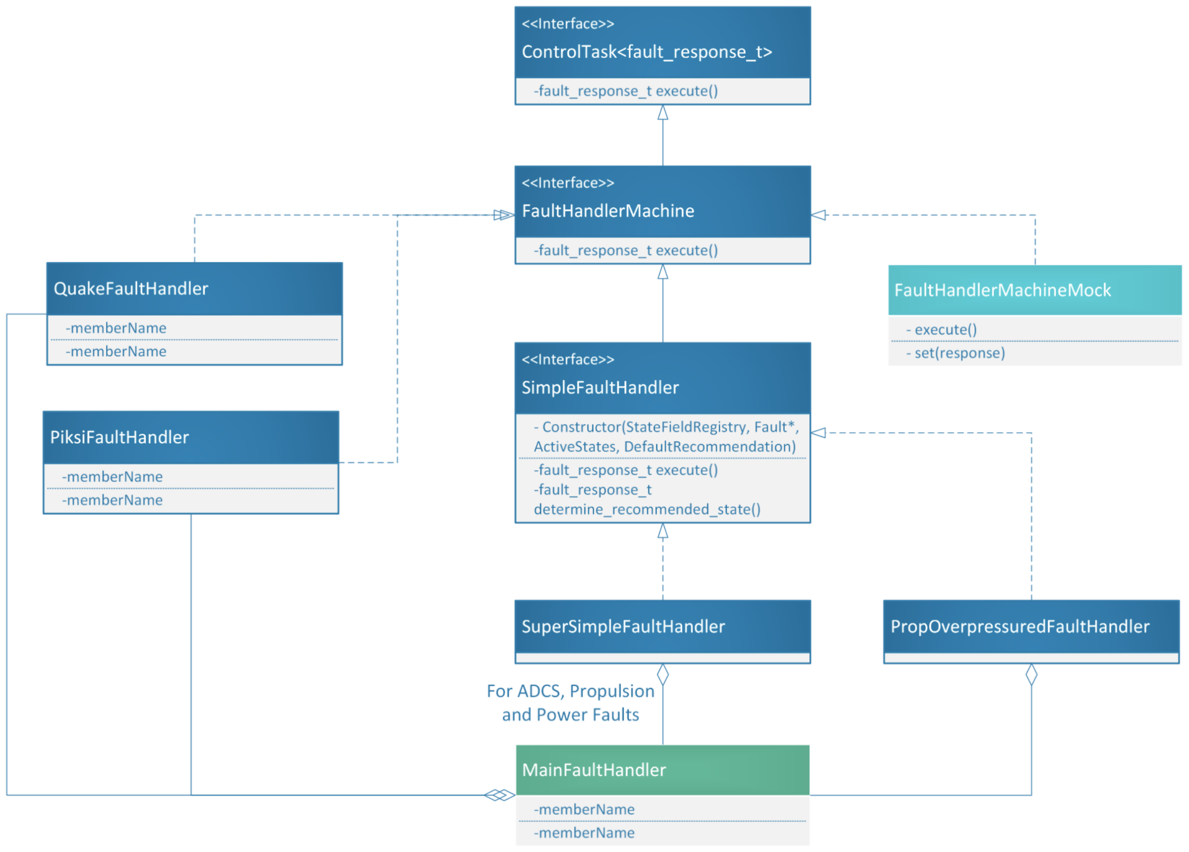

<<Interface>>
ControlTask<fault_response_t>
-fault_response_t execute()
<<Interface>>
FaultHandlerMachine
-fault_response_t execute()
QuakeFaultHandler
-memberName
-memberName
FaultHandlerMachineMock
- execute()
- set(response)
<<Interface>>
SimpleFaultHandler
- Constructor(StateFieldRegistry, Fault*, ActiveStates, DefaultRecommendation)
-fault_response_t execute()
-fault_response_t determine_recommended_state()
PiksiFaultHandler
-memberName
-memberName
SuperSimpleFaultHandler
PropOverpressuredFaultHandler
For ADCS, Propulsion and Power Faults
MainFaultHandler
-memberName
-memberName

Fig. 1: Diagram depicting the fault handler machine classes.

The core of all of these tasks is that they return a *fault_response_t*, which has value either *none*, *standby*, or *safehold*, corresponding to each possible recommended mission state by a fault handler. *QuakeFaultMachine* and *PiksiFaultMachine* implement this interface along with an internal state machine to keep track of their fault state, as described above. *MainFaultHandler* implements this interface as well.

For responding to simple fault detections, such as those captured by the Fault class, we've created the *SimpleFaultHandler* interface. It accepts a pointer to a Fault object, a set of mission states during which the fault handler responds to the fault (which we call *active states*), and the fault handler's recommendation upon detecting a fault.

The *SimpleFaultHandler* is an interface because it leaves its execute() function undefined, choosing instead to implement a function called *determine_recommended_state()*. This function produces the fault handler's recommended mission state as a function of the current mission state, the fault handler's "active states", and the state of the fault object. Subclasses of *SimpleFaultHandler* implement subsystem-specific actions in response to the fault. *SuperSimpleFaultHandler*, for example, implements no subsystem-specific response. *PropOverpressuredFaultHandler*, on the other hand, is a specialization that causes the propulsion tank to vent its propellant in order to relieve the pressure.

As depicted, the MainFaultHandler contains a few *SuperSimpleFaultHandler‘s (for ADCS, power, and propulsion failure-to-pressurize faults), a ‘PropOverpressuredFaultHandler*, a *QuakeFaultHandler*, and a *PiksiFaultHandler*. It runs all of its sub-tasks in order to administer the satellite’s fault response.

For the purpose of unit testing, we also have a *FaultHandlerMachineMock* that a tester can use to manually inject any kind of recommended mission state. It executes no subsystem-specific actions.

1.2.7 Flight Software Subsystems

Persistent State

In any computer system, it is important to keep track of persistent state that should survive expected or unexpected reboots of the system. For our system, the EEPROM is the only reliable means of state-saving, and we have an `EEPROMController` control task that manages the saving of certain state fields to the EEPROM.

Fields written to the EEPROM can either be signed/unsigned ints/chars, or booleans. This makes it easy to serialize or deserialize their values in and out of EEPROM. The list of EEPROM-saved fields can be found by running the Telemetry Info Generator (TIG); the “eeprom_saved_fields” key inside the produced JSON file by the telemetry info generator lists the set of state fields and their **saving period**, i.e. the number of cycles between queries of their value to save to the EEPROM.

The EEPROM is emulated in HOOTL via a JSON file that is stored on-disk.

Special Considerations

One of the fields saved to the EEPROM, `pan.state`, is special since it is a record of the current mission state.

Since it is only safe for the spacecraft to be in the standby, leader, follower, or close approach states if it has already gone through the startup and detumble states, the EEPROM controller ignores the saved value of `pan.state` if it is not startup, docked or docking. The reason docked and docking are allowed is because in those states the spacecraft is stable by virtue of its connection (or near-connection with the other spacecraft), and it would in fact be unsafe for the spacecraft to try to detumble.

Power Management

Our satellite runs on the Gomspace NanoPower P31U battery + management unit. It’s an space-rated system built for CubeSats and has an I2C interface that provides nearly complete control over all aspects of the Gomspace’s function. We employ two primary functions: checking the state-of-charge of our batteries and solar power systems, and power-cycling the Gomspace’s programmable outputs when desired.

Configuring Battery Settings

The Gomspace allows us to configure settings of the battery by sending a *config packet*. The config packet can hold settings for the following:

- The power point tracking mode
- Mode for battheater (0 = Manual, 1 = Auto)
- Battery heater low threshold. The battery will turn the heater on if the temperature goes below this value.
- Battery heater high threshold. The battery will turn the heater off if the temperature goes above this value.

- Nominal mode output value
- Safe mode output value
- Power point tracking mode for the boost converters

Reading State

The Gomspace provides a *housekeeping packet* of data that is requested by the satellite on every control cycle. This housekeeping packet contains the following important elements:

- The battery voltage
- The total system current output
- The total current into the system via the solar panels
- The boost converter voltages. These are the voltages set across the solar panels by the MPPT (maximum power point tracking) system, which adjusts the voltages such that the total power generated by the solar panels (which is voltage x current) is maximized.
- Output values (on or off), one for each of the six programmable outputs on the Gomspace.
- Time until power on/off
- Time remaining on the I2C, GND, and CSP watchdog timers. If this time gets close to zero, the Gomspace automatically resets power on the entire spacecraft in order to try to re-establish Flight Software and Gomspace communication. The intent of this feature is to shake flight software out of any infinite loops or fault conditions that might be causing it to stall.
- Number of I2C, GND, and CSP reboots
- Number of reboots of the Gomspace system (and by extension, the entire satellite)
- Cause of the last EPS reset
- Mode of the battery (0 = initial, 1 = undervoltage, 2 = safemode, 3= nominal, 4=full)
- The power-point tracking (PPT) mode (1=maximum power point tracking, 2=fixed). During flight, the PPT mode should be set to 1. While charging the battery, the mode should be set to 2.
- Temperature of boost converters and onboard battery
- The status of the battery heater (on or off)

Commanding

The gomspace controller allows us to set the photovoltaic (PV) input voltages, the outputs, the heater, and the power point mode from the ground. We can also reboot the satellite and reset the WDT counters from the ground using statefield commands. The gomspace will only set PV voltages when the Power Point Tracking mode is in `Fixed`.

Power Cycling

We can power cycle (i.e turn off and then on again) each of the six output channels of the Gomspace battery from the ground by setting statefield commands (e.g `gomspace.power_cycle_output1_cmd`) to `true`. Power cycling takes approximately 30 seconds to complete. It can be useful as a fault response. For example, the Quake Fault Handler powercycles the output channel connected to the radio up to three times during communication blackout periods.

DC-DC Converter

A 7-24V DC-DC converter is connected to the Gompace battery and powers the ADCS system, the propulsion system, and the docking system. The ADCS system is connected to the `ADCSMotorDCDC_EN` pin. The propulsion and docking systems are connected to the `SpikeDockDCDC_EN` pin. These pins can be enabled, disabled, and reset from the ground using statefield commands. The DC-DC pins should not be disabled except in exceptional circumstances, such a failure in I2C communication.

Attitude Determination and Control

The attitude determination and control subsystem (ADCS) box of the satellite is a 0.5U unit attached at the end of our spacecraft opposite its docking face. It has its own microcontroller, a Teensy 3.6, that runs a specialized software and is connected to the Flight Controller Teensy over I2C. The ADCS box provides fairly robust but slow pointing control–the slowness is sufficient for PAN's purposes.

Attitude Determination System

The attitude on the spacecraft is determined using a combination of three sensors: the sun sensors, the gyroscope, and the two magnetometers.

Sun Sensors

The sun sensors are 5 arrays of 4 sensors, with each array attached to one of the faces of the ADCS box, and each sensor in the array angled off the surface of the box in a different way. The sun sensors are nothing more than phototransistors attached to an analog-to-digital converter. The five analog-to-digital converters (one for each array) are connected via I2C to the Teensy.

The current through each phototransistor can be read as a voltage on the Teensy. The currents on all twenty sensors are fed through a precomputed linear regression to determine the vector to the sun relative to the spacecraft in its body frame. If such a regression returns inconclusive results then the ADCS Teensy lets the Flight Controller know (see *Interface with Flight Software* below).

Gyroscope and Heater

The gyroscope is connected over I2C to the ADCS Teensy. To ensure the gyroscope measurement does not drift due to thermal fluctuations, the gyroscope is embedded underneath a resistive heating device that operates via a bang-bang controller. The setpoint of the controller is managed by the Flight Controller.

Magnetometers

There are two magnetometers on the spacecraft.

MAG1: Todo: details about MAG1

MAG2: Todo: details about MAG2

Each magnetometer can be individually commanded between `IMU_MAG_NORMAL` and `IMU_MAG_CALIBRATION` modes. Data is polled from both magnetometers simultaneously.

Attitude Control System

We achieve attitude control via 3 reaction wheels and 3 magnetorquers, one for each axis of the spacecraft. The wheels are controlled via “ramp command”, which sets their angular acceleration and thus provides torque-based control over the spacecraft’s attitude. The magnetorquers are provided a magnetic moment command, via which they can execute a torque on the spacecraft.

The ADCS Software

TODO explain ADCS Software functionality.

Interface with Flight Software

The interface of the ADCS box Teensy with the flight Teensy is over register-based I2C, effectively SMBUS. In this relationship the ADCS Teensy is a slave and the flight Teensy is a master.

Reading from ADCS Controller

The flight controller can read values off of the ADCS Teensy via a “point-and-read” interface. The flight controller first sets the value of a read pointer, which specifies the register address at which it wants to receive values. The ADCS Teensy then responds with a set of values that begin at the register specified by the flight controller and run contiguously up to some other register address. This allows the flight controller to read values off of the ADCS controller in bulk, which reduces protocol overhead when accessing related values.

Below we list the “read registers” on the ADCS and where a read operation ends when the read pointer is set to that register address. As an example for explaining the previous paragraph, note that setting the read pointer to the X-value of the magnetometer causes the ADCS Teensy to report back the X, Y, *and* Z values of the magnetometer. This is sensible since any control scheme would want all three values off of the ADCS device.

TODO insert table from Kyle’s document

Writing to ADCS Controller In order to actuate attitude commands, the ADCS box provides registers that can be written to. This list of registers is specified below.

TODO insert table from Kyle’s document

ADCS Hardware Availability Table (HAVT)

For every `Device` connected to the ADCS Teensy, the Teensy tracks the “functionality” state of the `Device`. If it is disabled, then if the device is an actuator, no actuations will be performed. If the `Device` is an `I2CDevice`, then no I2C transactions will occur with that device.

Internally within ADCS Software, there is a Hardware Availablility Table that reports the funct ionality state of each device. I2C devices automatically disable (set their functionality state to false) themselves if too many consecutive I2C transactions fail. All devices are initially enabled ADCS Teensy boot, so if all `setup()` calls are succesful, the initial HAVT table will be represented by a bitset of all `1`’s.

The ADCS Teensy has a read register dedicated to reading the state of the HAVT table. There are two seperate command registers to intewith the HAVT table, one for commanding `reset()`’s, and another for commanding `disable()`’s. All three are represented as a 32 bit long bitsets.

On every ADCS cycle, the ADCS Teensy will actuate a `reset()` or `disable()` if the index of the command table corresponding to a device has a `1` in that position. Therefore, nominally the reset and disable command registers are all commanded as `0`’s from the FSW Teensy.

Flight Software Components for ADCS

Several control tasks manage the ADCS system. They are: the ADCS box monitor, the ADCS attitude estimator, the ADCS computer, the ADCS commander, and the ADCS box controller.

- The ADCS Box Monitor and Controller are basic device-interface control tasks that do the simple job of reading sensor values and writing wheel and magnetometer commands to the ADCS peripheral.
  - ADCS Box Monitor-specific behaviors:
    - If a sensor reading is out of bounds, ADCSBoxMonitor will set a corresponding flag as true. Otherwise, it is set to false.
    - After reading the ADCS HAVT table, ADCSBoxMonitor will `signal()` a corresponding fault if any of the wheels, or the wheel potentiometer report as not functional. Otherwise, the flag is `unsignal()`'ed.
  - ADCS Box Controller-specific behaviors:
    - Renews the calculation of the sun vector if **ADCSMonitor** reported that a previous calculation was no longer in progress.
    - Applies the desired HAVT reset or HAVT disable vectors to the ADCS box.
- The ADCS attitude estimator takes inputs from the box monitor to produce a filtered estimate of the spacecraft's attitude.

  TODO: What happens when inputs are NaN?
- The ADCS computer, using the high-level ADCS strategy dictated by the mission manager, creates a desired attitude for the spacecraft.

  The desired attitude is provided via four vectors: a "primary" pointing objective; the body vector that should be aligned with the primary pointing objective; and the "second" pointing objective and body vector.
- The ADCS commander implements a control law to convert the desired attitude and rate into wheel and magnetorquer commands for the spacecraft.
  - If the `adcs_state` is in startup, this control task sets the ADCS box to passive mode which disables all actuation (magnetorquers and wheels) regardless of the MTR and Wheel commands coming from **ADCSCommander**. In all other `adcs_states` ADCSController will dump all the desired commands from **ADCSCommander** into the ADCS box using the ADCS Driver.

TODO insert state field names

Propulsion System Management

The purpose of this document is to detail the Propulsion Controller Subsystem of Flight Software. The Propulsion System is responsible for generating thrust in order to accelerate the satellite. It is also responsible for monitoring the state of the propulsion system hardware and handling detected faults.

The software components of the Propulsion System consists of the Propulsion System driver, the Propulsion System Controller, and the Propulsion System Fault Handler.

The hardware components of the Propulsion System consists of the inner tank, the outer tank, the inner tank temperature sensor, the outer tank temperature sensor, the outer tank pressure sensor, the two intertank valves, and the four outer tank valves.

This document is split into four main sections. The first section gives a high-level overview of the Propulsion System, its components, and the responsiblities of each component. The second section details the Propulsion System state machine. The third section details the Propulsion System driver. The last section consists of operational notes, warnings, and known issues.

Overview

This section gives and overview of the components of the Propulsion System, their responsibilities, and defines terminology used in the rest of this document.

Tank1

`Tank1`, the inner tank, is responsible for pressurizing `Tank2`. At the start of the mission, `Tank1` is filled with liquid propellant. The propellant is released into `Tank2` through one of the two intertank valves, which causes pressure to build up in Tank2. The two valves on `Tank1` are referred to as the `primary valve` and the `backup valve`. `Tank1` also has a temperature sensor, which is used to detect and handle faults.

Tank2

`Tank2`, the outer tank, is responsible for accelerating the satellite. It consists of four valves arranged tetrahedrally: `valve1`, `valve2`, `valve3`, `valve4`. Each valve is assigned a schedule, and the four valve schedules along with a firing time consist of a *firing schedule*. The direction of acceleration is, therefore, determined by the firing schedule. `Tank2` has a temperature sensor and a pressure sensor. The pressure sensor on `Tank2` is used to indicate when `Tank1` should stop pressurizing `Tank2`. Both the pressure sensor and temperature sensor are used to detect and handle faults.

Firing Schedule

The firing time is determined by `cycles_until_firing`, which is the number of control cycles from the current control cycle at which the valves shall fire. It is defined relative to the current control cycle. For example, if the current control cycle is 13, then `cycles_until_firing` of 8 means that the valves shall fire when the satellite is in control cycle 21.

The `valve schedules` are in units of milliseconds with a maximum value of 1000. When the satellite enters the control cycle specified by the firing time, the valves will open for the duration of their assigned schedules.

The Propulsion System Controller will only execute what it considers a `valid firing schedule`. Any schedule considered invalid will be ignored.

If any valve is assigned a schedule greater than 1000, then that entire firing schedule is invalid. If the Propulsion System Controller believes that it will not have enough time to pressurize `Tank2` by the desired firing time cycle, then this schedule will also be considered invalid. A valid firing schedule is, therefore, a firing schedule in which all valve schedules are no greater than 1000 ms and the firing time is far enough into the future that the Propulsion System has time to pressurize.

Propulsion Controller State Machine

This section details the Propulsion Controller (`PropController`), which is implemented as a state machine. The state machine interacts with the propulsion system via the propulsion system driver.

The Propulsion Controller is defined in `PropController.hpp` and implemented in `PropController.cpp`.

The firing time is determined by `prop.cycles_until_firing`. It is time to fire, when `prop.cycles_until_firing` is 0.

There are techncially two copies of the firing schedule: the state machine schedule and the driver schedule. The state machine schedule consists of the following statefields: `prop.sched_valve1`, `prop.sched_valve2`, `prop.sched_valve3`, and `prop.sched_valve4`. The values in this state field are copied to the driver's schedule one cycle prior to the firing time.

Propulsion Controller States

This section details state transitions and entry conditions (preconditions) of the states in the Propulsion System state machine.

Disabled

- In this state, propulsion system will defer decisions to the ground (or other subsystems) and will only read the sensor values
- No transitions are possible from this state
- There are no entry conditions; any state may enter enter this state.

Idle

- In this state, the propulsion system is ready to process and execute firing schedules
- Transitions to **handling fault** if any hardware fault is faulted (has persistently been signaled)
- Transitions to **await pressurizing** or **pressurizing** upon reading a valid schedule
- To enter this state, `DCDC::SpikeDockDCDC_EN` pin must be `HIGH`

Await Pressurizing

- In this state, the state machine has accepted the current schedule but has decided to wait until it is closer to the firing time before starting to pressurize
- Transitions to **handling fault** if any hardware fault is faulted (has persistently been signaled)
- Transitions to **pressurizing** if it meets the entry conditions for **pressurizing**
- To enter this state, the current state must be **idle**, the schedule must be valid, and there must be more than enough time to pressurize

Pressurizing

- In this state, propulsion system is currently pressurizing
- Transitions to **handling fault** if any hardware fault is faulted (has persistently been signaled)
- If `Tank2` pressure reaches `prop.threshold_firing_pressure`, then transition to **await firing**
- If `Tank2` pressure fails to reach the `prop.threshold_firing_pressure` within `prop.max_pressurizing_cycles` and the `prop.pressurize_fail` has not been suppressed, then transition to **handling fault**.
- If `Tank2` pressure fails to reach the `prop.threshold_firing_pressure` within `prop.max_pressurizing_cycles` and the `prop.pressurize_fail` has not been suppressed, then transition to **await firing**.
- If the schedule is no longer valid, transition to **disabled**
- To enter this state, the current state must be either **await pressurizing** or **idle** and there must be exactly `min_cycles_needed()-1` cycles until it is time to fire

Await Firing

- In this state, propulsion system has reached threshold pressure and will remain in this state until it is time to fire
- Transitions to **handling fault** if any hardware fault is faulted (has persistently been signaled)
- Transitions to **firing** when `prop.cycles_until_firing` is 0. On this cycle, the values of the firing schedule in the statefields will be copied to the schedule in the propulsion system driver.
- To enter this state, the current state must be **pressurizing** and the schedule must be valid

Firing

- Transitions to **handling fault** if any hardware fault is faulted (has persistently been signaled)
- On each cycle, copies the values of the driver firing schedule into the state machine firing schedule
- Transitions to **idle** when all values of the firing schedule is 0
- To enter this state, the current state must be **await firing** and `prop.cycles_until_firing` must be 0

Handling Fault

- To enter this state, at least one of `prop.pressurize_fail`, `prop.overpressured`, `prop.tank2_temp_high`, `prop.tank1_temp_high` is faulted
- Transitions to **venting** if the entry conditions of **venting** are meets
- Transitions to **idle** if no fault is faulted

Venting

- In this state, faults relating to overpressure or high temperatues have been detected for several consecutive control cycles
- To enter this state, at least one of `prop.overpressured`, `prop.tank2_temp_high`, `prop.tank1_temp_high` is faulted
- Transitions to **disabled** if after executing `prop.max_venting_cycles` number of venting cycles, the fault in question is still faulted
- If faults are faulted for both `Tank1` and `Tank2` at the same time, then the `PropFaultHandler` will coordinate the venting protocol to make the tanks take turn venting.
- If venting `Tank1` or `Tank2` due to high temperatures, transition to **idle** if the temperature falls below `max_safe_temp` (48 C)
- If venting `Tank2` due to high pressure, transition to **idle** if pressure falls below `max_safe_pressure` (75 psi)
- See the PropFaultHandler section below for details on the venting protocol

Pressurizing Protocol

The pressurizing protocol consists of executing a sequence of *pressurizing cycles* up to a maximum of `prop.max_pressurizing_cycles` pressurizing cycles. A pressurizing cycle consists of *filling* period and a *cooling* period. The filling period is given by `prop.ctrl_cycles_per_filling` and the cooling period is given by `prop.ctrl_cycles_per_cooling`.

Therefore, in a single pressurizing cycle, a valve on `Tank1`, given by `prop.tank1.valve_choice` is opened for `prop.ctrl_cycles_per_filling` number of control cycles and then closed for `prop.ctrl_cycles_per_cooling` number of control cycles. At each control cycle, `Tank2` pressure, given by `prop.`

`tank2.pressure`, is compared with `prop.threshold_firing_pressure`. If `Tank2` pressure reaches the threshold firing pressure, then the state machine transitions to **firing**.

If after `prop.max_pressurizing_cycles`, the pressure of `Tank2` has not reached the threshold firing pressure, then the `prop.pressurize_fail` fault is signaled. This fault has a persistence of 0, so if it has not been previously suppressed by the ground, the state machine will transition to **handling fault**.

If it has been suppressed by the ground, the state machine will transition to **await firing**.

Interface

The only method that is particularly useful to other subsystems is `min_cycles_needed()`. The rest are documented here solely because they are public.

`min_cycles_needed()` Returns the minimum number of control cycles needed for a schedule to be accepted. If a schedule is accepted, the state machine transitions from **idle** to **await firing**.

`is_at_threshold_pressure()` Returns true if `Tank2` pressure has reached the threshold firing pressure

`is_tank2_overpressured()` Returns true if `Tank2` pressure has exceeded `max_safe_pressure`

`is_tank1_temp_high()` Returns true if `Tank1` temperature has exceeded `max_safe_temp`

`is_tank2_temp_high()` Returns true if `Tank2` temperature has exceeded `max_safe_temp`

`check_current_state(prop_state_t expected)` Returns true if the current state is the expected state

`can_enter_state(prop_state_t desired_state)` Returns true if the state machine can enter the desired state from its current state

`write_tank2_schedule()` Copies the state machine firing schedule from the statefields to the propulsion system driver schedule

State Fields

prop.state The current state of the state machine (values defined in `prop_state_t.enum`)

prop.cycles_until_firing Determines the firing time relative to the current control cycle count

prop.sched_valve1 The schedule for `Tank2 valve 1` in milliseconds

prop.sched_valve2 The schedule for `Tank2 valve 2` in milliseconds

prop.sched_valve3 The schedule for `Tank2 valve 3` in milliseconds

prop.sched_valve4 The schedule for `Tank2 valve 4` in milliseconds

prop.max_venting_cycles The maximum number of venting cycles to attempt before disabling the propulsion system

prop.ctrl_cycles_per_closing The number of control cycles to wait between opening valves during a venting cycle (default 1 second worth of control cycles)

prop.max_pressurizing_cycles The maximum number of pressurizing cycles to attempt before transitioning to **handling fault**

prop.threshold_firing_pressure The minimum pressure needed in `Tank2` to execute a firing schedule

prop.ctrl_cycles_per_filling The number of control cycles to open the `Tank1` valve during a pressurizing cycle (default 1 second worth of control cycles)

prop.ctrl_cycles_per_cooling The number of control cycles to wait between opening a `Tank1` valve during a pressurizing cycle (default 10 seconds worth of control cycles)

prop.tank1.valve_choice Specifies the `Tank1` valve that will be opened during pressurizing or venting cycles (default is 0 for the `primary valve`)

prop.tank2.pressure The current pressure of `Tank2` given by its pressure sensor

prop.tank2.temp The current pressure of `Tank2` given by its temperature sensor

prop.tank1.temp The current pressure of `Tank1` given by its temperature sensor

prop.pressurize_fail Fault field indicating that the state machine has executed `prop.max_pressurizing_cycles` and has still failed to reach `prop.threshold_firing_pressure`

prop.overpressured Fault field indicating that the pressure in `Tank2` exceeds `max_safe_pressure` (75 psi)

prop.tank1_temp_high Fault field indicating that the temperature in `Tank1` exceeds `max_safe_temp` (48 C)

prop.tank2_temp_high Fault field indicating that the temperature in `Tank2` exceeds `max_safe_temp` (48 C)

Propulsion System Fault Handler

The Propulsion System Fault Handler is defined in `PropFaultHandler.h` and implemented in `PropFaultHandler.cpp`. It is only active when the `prop_state` is in **venting** or in **handling fault**.

Four possible faults have been defined by the Propulsion Subsystem: `prop.pressurize_fail`, `prop.overpressured`, `prop.tank2_temp_high`, `prop.tank1_temp_high`. Handling `prop.pressurize_fail` is deferred to the ground. The state machine will attempt to resolve the other three faults in the **venting** state.

Venting Protocols

The protocol for venting one tank is similar to the the protocol for pressurizing. The maximum number of venting cycles is given by `prop.max_venting_cycles`. The number of control cycles to open a valve is given by `prop.ctrl_cycles_per_filling`.

Venting `Tank1` is almost the same as pressurizing except that the period between opening the valve has been shorten to `prop.ctrl_cycles_per_closing` instead of `prop.ctrl_cycles_per_cooling`.

Venting `Tank2` is the same as venting `Tank1` except the state machine will open a different valve from `Tank2` after each venting cycle. Whereas `Tank1` always vents through `prop.tank1.valve_choice`, `Tank2` will cycle through its four valves.

The state machine leaves the **venting** state when the fault(s) associated with the tank that it is currently venting are no longer faulted.

When faults are active from both tanks indicating that the state machine should vent both tanks, the `PropFaultHandler` is responsible for making the tanks take turns venting. `PropFaultHandler` will save the current value of `prop.max_venting_cycles` and then set `prop.max_venting_cycles` to 1. This will cause the venting cycle to end after 1 cycle and transition unconditionally to **handling fault**. `PropFaultHandler` will then be responsible for counting the number of venting cycles executed. It will consider a single venting cycle to consist of venting both `Tank1` and `Tank2` for one venting cycle each.

Should one of the faults become unsignaled during this protocol, `PropFaultHandler` will restore the old value of `prop.max_venting_cycles` and the continue to vent if necessary.

Propulsion System Driver

This section details the purpose of the propulsion system driver, its components, and its public interface.

The driver is responsible for opening and closing valves on both tanks and executing the firing schedule. The protocols for validating the firing schedule and executing the pressurizing and venting operations are left to the `PropController`.

The Propulsion System Driver is defined in `PropulsionSystem.hpp` and implemented in `PropulsionSystem.cpp`. It consists of three singleton (static) objects: `PropulsionSystem`, `Tank1`, and `Tank2`. The objects are globally accessible, but subsystems are advised to not directly interact with these objects. The public interface is documented here for completion.

The two `Tank1` valves are indexed (`valve_idx`) at 0 and 1. The four `Tank2` valves are indexed at 0, 1, 2, and 3.

Interface

`PropulsionSystem.is_functional()` Returns true if the Propulsion System is operational (i.e. able to execute firing schedules and read sensors).

`Tank1.get_temp()` Returns the temperature sensor reading for `Tank1` in degrees Celcius.

`Tank2.get_temp()` Returns the temperature sensor reading for `Tank2` in degrees Celcius.

`Tank2.get_pressure()` Returns the pressure sensor reading for `Tank2` in psi.

`Tank1.is_valve_open(valve_idx)` Returns true if the Tank1 valve at `valve_idx` is opened

`Tank2.is_valve_open(valve_idx)` Returns true if the `Tank2` valve at `valve_idx` is opened

`PropulsionSystem.set_schedule(valve1, valve2, valve3, valve4)` Sets the firing schedule for the four `Tank2` valves

`PropulsionSystem.reset()` Shuts off all the valves in both `Tank1` and `Tank2` and clears the firing schedule

`PropulsionSystem.start_firing()` Executes the firing schedule immediately

`PropulsionSystem.disable()` Ends the firing schedule regardless of whether the entirety of the firing schedule has been executed

`PropulsionSystem.open_valve(tank, valve_idx)` Opens the valve at `valve_idx` for `tank`

`PropulsionSystem.open_valve(tank, valve_idx)` Closes the valve at `valve_idx` for `tank`

Implementation Notes

When `start_firing()` is called, an interrupt timer will cause an interrupt every 3ms. The interrupt handler is responsible for opening the valves for the duration of the assigned schedules and closing the valves when they are within 10ms of completing their schedules. The interrupt timer is disabled by calling `PropulsionSystem.disable()`.

While the interrupt timer is enabled, the schedule may not be modified in any way.

Calling `PropulsionSystem.reset()` implicitly calls `PropulsionSystem.disable()`.

Operational Notes

The preconditions for entering a state can be bypassed by manually setting prop_state to the desired state.

This is because `can_enter()` is only evaluated when the state machine itself is attempting to transition states. Be warned that it may be possible for the state machine to be indefinitely stuck in a state since it may only transition to a new state if it meets that new state's preconditions.

To make a firing occur immediately, set the firing schedule and transition to await_firing

To force the state machine to immediately execute a schedule, set `prop.cycles_until_firing` to 0 and set `prop_state` to **await firing**. This will cause the firing schedule to immediately be copied into the driver and executed on the next control cycle. Note that each of the valve schedules must still be no greater than 1000 ms, otherwise, the driver will ignore the entire firing schedule.

Do not manually set prop_state to firing.

Manually setting `prop_state` to **firing** is counterproductive and will not cause the schedule to be executed. This is so because the call to `PropulsionSystem.start_firing()` occurs in the entrance protocol of the **firing** state, which can only be executed when transitioning from **await firing**.

Do not manually set prop_state to a state other than disabled while it is pressurizing or venting.

Manually setting `prop_state` to **disabled** can be safely done from any state. It is, however, not advisable to manually set `prop_state` to a state other than **disabled** while it is in the **pressurizing** or the **venting** state. The reason for this is because valves are manually opened by the `PropController`. If the state machine is interrupted while the valves are opened, the `PropController` will not get the opportunity to close these valves.

The Propulsion System does not work if DCDC Spike and Hold pin is not enabled.

The Propulsion System requires that `DCDC::SpikeDockDCDC_EN` pin be high. The state machine will still execute if the pin is not high, but its behavior is undefined. The state machine will likely erroneously detect faults.

The Propulsion System will close the valve when fewer than 10ms remain on its schedule.

For example, if a `Tank2` valve is scheduled to fire for 200 ms, then it is guaranteed to open for at least 190 ms but no more than 200 ms. Once firing, schedules are checked every 3 ms. Therefore, all schedules under 10ms will be considered valid by the state machine but will not be executed by the Propulsion System driver.

A schedule can technically be cancelled at any time before the scheduled firing time.

The state machine does not provide any convenient way to accomplish this. If a subsystem wishes to cancel a firing schedule, then it may do so as long as `prop.cycles_until_firing` is not 0. The subsystem can set prop_state to **idle** and invalidate the schedule by clearing `prop.cycles_until_firing`. Similarly, if the subsystem would like to replace the schedule with a different schedule, then that subsystem should write the schedule to the appropriate state fields and then manually set `prop_state` to **idle**.

Setting prop_state to disabled will not clear the firing schedule.

A subsystem can therefore pause or delay the schedule by setting `prop_state` to **disabled**. Since the firing time is relative to the current control cycle, a firing schedule that is valid prior to disabling the state machine will still be valid should the subsystem set `prop_state` to the state it was in prior to being **disabled**.

Known Issues

When testing the Propulsion System and running multiple tests within a single process, it does not matter that the `registry` or the `TestFixture` is destroyed between tests. Since the objects are static, the results of previous tests will always persist, so to avoid strange test results, the `TestFixture` should reset the fields of the static objects.

Guidance, Navigation, and Control

Attitude Determination

TODO

Attitude Control

TODO

Orbit Estimator

Author: Nathan Zimmerberg (nhz2@cornell.edu) Started: 13 May 2020 Last Updated 14 May 2020

Main Goal

The goal of the orbit estimator is to estimate the position and velocity of self and the target satellites given sensor inputs and messages sent from ground.

Input

GPS: Each satellite has a Piksi GPS receiver (see *GPS Receiver*).

Ground Messages: Ground software will uplink the target's estimated orbit whenever possible.

Ground software can also modify the orbit estimator, including resetting, changing the propagation and sensor models, and manually setting the orbit.

Propulsion: The last thruster firing.

Attitude Determination and Control: The estimated attitude, combined with the last thruster firing to get the change in velocity in ECEF.

Output

Expected self and target Orbit, statistics, and debug information.

Software Components

The following components need to be finished and tested before getting combined into the main orbit estimator.

Check Orbit Validity: Done Check Orbits are in low earth orbit, this is useful for catching filter instabilities.

A valid orbit has finite and real position and velocity, is in low earth orbit, and has a reasonable time stamp within MAXGPSTIME_NS, MINGPSTIME_NS.

Low earth orbit is a Orbit that stays between MINORBITRADIUS and MAXORBITRADIUS.

Short Orbit Propagation with Jacobian Output: Done The EKFs need to propagate the state and covariance in between GPS readings using this.

Orbit GroundPropagator: Done Class to propagate orbits sent from ground.

The GroundPropagator tries to:

1. minimize the number of grav calls needed to get an up to date orbit estimate.
2. use the most recently input Orbit.

Implimentation details:

Under normal conditions, this estimator just propagates the most recently uplinked Orbit. in current.

If a new Orbit get uplinked it will normally get put in catching_up, and the estimator will propagate it to the current time in the background while still propagating current as the best estimate. Once catching_up is done propagating it replaces current.

If another Orbit gets uplinked while catching_up is still being propagated in the background, it gets stored in to_catch_up. This ensures too many Orbits getting uplinked won't overload the estimator and prevent it from making progress. If to_catch_up takes fewer grav calls to finish propagating than catching_up it replaces catching_up. Also to_catch_up replaces catching_up if catching_up finishes propagating.

Propagator details: High order integrators Yoshida coefficients from: https://doi.org/10.1016/0375-9601(90)90092-3 The higher order propagator step right now works like this, first it converts position and velocity in ecef to relative inertial coordinates to a close reference circular orbit. Then it does a series of drift-kick-drift steps (see https://en.wikipedia.org/wiki/Leapfrog_integration) where a drift is rel_r= rel_r+rel_v*dt*0.5; and a kick is rel_v= rel_v + g_ecef0*dt; For the low order step(2nd ish) there is just one drift-kick-drift, for the higher order step(6th ish) Yoshida coefficients are used to do 7 drift-kick-drifts with a series of d*dt: Where somehow this magical series of time steps cause some errors to cancel out. Finally when the step(s) are done the relative position and velocity are converted back to ecef.

Single Orbit Extended Kalman Filter: WIP Use an extended Kalman filter to estimate the self orbit from GPS data.

This is currently implemented only in MATLAB, but the current implementation is too computationally expensive and numerically unstable to be directly used in flight software. I am working on a square root Kalman filter, and more carefully managing the computational load for the C++ version.

Double Orbit Extended Kalman Filter: WIP Use an extended Kalman filter to estimate the self and target orbit from CDGPS and GPS data.

This is currently implemented only in MATLAB, but the current implementation is too computationally expensive and numerically unstable to be directly used in flight software. I am working on a square root Kalman filter, and more carefully managing the computational load for the C++ version.

Testing

Unit Tests: Unit tests are run in CI, and the teensy.

Unit tests check that the orbit propagation is accurate, and the Kalman filter math is right.

Estimator Performance Tests:

To test the orbit estimator performance I am using data from GRACE-FO and from PSIM.

The workflow is to generate a file of sensor data and truth on every control cycle from a full PSIM sim. Then open that file in a Jupyter Notebook and plot the performance of the estimator under test.

For example see https://github.com/pathfinder-for-autonomous-navigation/psim/blob/Jupyter-Notebook-Plotting-Utility/estimatortest/Orbit-estimator-test.ipynb

The C++ components can be easily wrapped in python using pybind11 and cppimport so tweaks to the C++ code can be quickly tested.

This is much faster than running a full PSIM sim and doesn't require access to MATLAB.

Also Jupyter Notebook can be run over SSH so if someone has an old laptop, can't get the code to compile, or doesn't have MATLAB, they can still visually test the estimator on a Linux server.

Orbital Control Algorithm

TODO

Telemetry Management

The flight software needs to be able to both send its data to the ground and have the capacity to accept commands from the ground. These tasks are handled by the `DownlinkProducer` and `UplinkConsumer` control task, respectively, which are described in more detail below.

Downlink Producer

The downlink producer has the responsibility of managing the selection of state fields that are downlinked to the ground. This is important since the downlink bandwidth is extremely limited, down to a size of at most 70 bytes per downlink packet, with low probability of communication in general.

Here are some definitions we lay down to begin with:

- A `downlink snapshot` represents a snapshot of all of the data we would ever want from the spacecraft at a particular time.
- A `downlink frame` is a collection of 70-byte `downlink packets` which contain the data of a snapshot.

To downlink fields in a manageable way, the state fields are partitioned into groups called *flows*. The fields in a flow are all transmitted together. This allows telemetry design to be thought in terms of "which flows do I want to send down?", as opposed to "which fields do I want to send down?", which makes management much easier.

Each flow has the following priorities: a *flow ID* that's unique to each flow, and a *flow priority*, ranging from -1 to the total number of flows. A flow priority of -1 means that the flow is inactive and not downlinked to the ground; a flow priority of 0 is the highest priority, and a flow priority of `number of flows - 1` is a flow with the lowest priority.

Flows are specified in a CSV file under *src/* in Flight Software. This CSV file is processed by a Python script to produce a compilable C++ file with the same data.

Serializing Flows to a Stream

When downlinking, the downlink producer arranges the active flows by priority in-place, and then writes each flow to the downlink frame in order. Writing a flow to the packet means writing the flow's ID, and then writing each of the serialized fields in the flow.

Since the set of flows might require more than 1 downlink packet, we mark each packet with a header bit (1 or 0) that indicates if the packet is the first packet in the downlink frame. This header bit is inserted while the flows are being written to the downlink frame; if a flow is going to cause the data to overflow into a new downlink packet, a header bit is written first, and then the remainder of the flow is written. The effect of this scheme is that if the header bits are removed from the downlink frame, the resulting data is a continuous stream of flows.

Uplink Consumer

In our system design, the ground is only allowed to send commands in the form of state field updates: behavior that the ground wants to modify has to be attached to a state field that can control that behavior. For example, one common ground task is to change the *Mission Management* state, which it can easily do by just setting the value of the mission state.

The structure of an uplink packet is thus a set of key-value pairs, with the key being the uplinked state field's *index* in the list of uplinkable state fields, and the value being the actual value of the state field. The index is compressed to the minimum number of bits required to specify the maximum possible index, and the value of the uplinked state field must be serialized using its serializer.

Uplink packets can be produced using the Uplink Producer utility, which is described in greater detail here.

The Radio Manager

The Quake Manager controls the satellite's radio and is responsible for managing the link layer of PAN's communications protocols (see *Telemetry Management*).

The Quake requires the teensy's serial receiving buffer to have atleast 512 bytes of capacity.

Docking System

The docking system has three components: the docking magnets, the docking motor, and the docking switch. The latter two components are controllable and readable by flight software.

Our docking at a range of 1 m happens passively via 4 strong neodymium magnets stored in the docking adapters of the two satellites. During nominal mission operation, the strong fields created by these magnets would wreak havoc with the onboard magnetometers, so they are nominally stowed in a quadrupole configuration so that they have very little net magnetic field past the docking face. This is achieved by having two magnets point N-S, and another, adjacent set of magnets point S-N.

The docking magnets are reconfigured from their quadrupole arrangement to a dipole arrangement via the docking motor, which is a stepper motor that turns one of the magnet pairs in the right direction for achieving the correct polarity.

During the docking operation, both satellites have a switch on their docking face that is depressed when the satellites dock with one another. The switch is how we detect that the actual docking operation occurred.

Operation of the Docking Motor

The magnets are initially in the docking configuration, and thus the docking motor is only necessary for undocking and redocking after the first successful docking of the two satellites has occured. There are four main statefields useful in checking and changing the state of the docking motor. These are *docksys.dock_config*, which is false when the magnets are in the undocked position and true when in docked position, *docksys.is_turning*, which is true if the motor is being signalled to step and false otherwise, *docksys.docked*, which is true if the docking switch is depressed and false otherwise, and *docksys.config_cmd*, the writeable state that tells the system what configuration the magnets should be in (true for docked, false for undocked).

Two additional writeable statefields exist, *docksys.step_angle* and *docksys.step_delay*. The step angle of the motor is constant based on its setup, but since the motor data sheet did not match observed values this step angle was calculated from experimental trials and so we leave it adjustable. The step delay determines how fast the motor is turning. The load from the magnets can vary based on position and changes to the setup from shaking during the mission, as can power from the battery. Thus, the step delay can be changed if the system is not successfully changing configuration as longer delays will yield more torque and control, although they are slower. The *docksys.step_delay* is useful in troubleshooting issues if the system does not successfully turn with the initial values.

To operate the docking motor, the *docksys.config_cmd* should be sent from the ground and the motor should turn into the desired position within a minute. If that does not work automatically, the *docksys.step_delay* value should be increased and the process repeated.

GPS Receiver

The PAN mission utilizes the Piksi, which is a Carrier Differential GPS (CDGPS). Two Piksi units detect the phase difference between their GPS signals, and use that phase difference to estimate a relative difference in position, called the baseline vector. This position difference vector is accurate to 1cm which enables docking through cdGPS navigation alone for PAN.

Time Propagation

TODO

Piksi Control Task

The Piksi Control Task or PiksiControlTask requires that the serial buffer that is connected to the flight computer have a receiving buffer capacity of 1024 bytes. This is because within a 120 ms control cycle, there can potentially be two piksi packets. Each packet nominally contains about 299 bytes. 512 is not sufficient, but the next power of two, 1024 is sufficient to contain 299 * 2.

Data

Piksi Velocity readings are in ECEF coordinates and in millimeters persecond

Piksi Position readings are in ECEF coordinates and in meters

Piksi Baseline readings are in ECEF coordinates and in millimeters

1.2.8 PTest Documentation

PTest is an extremely robust testing architecture that tests our flight software in conjunction with our mission simulation in both a Hardware-out-of-the-Loop (HOOTL) and Hardware-in-the-Loop (HITL) configuration. The HITL tests can be further broken down into

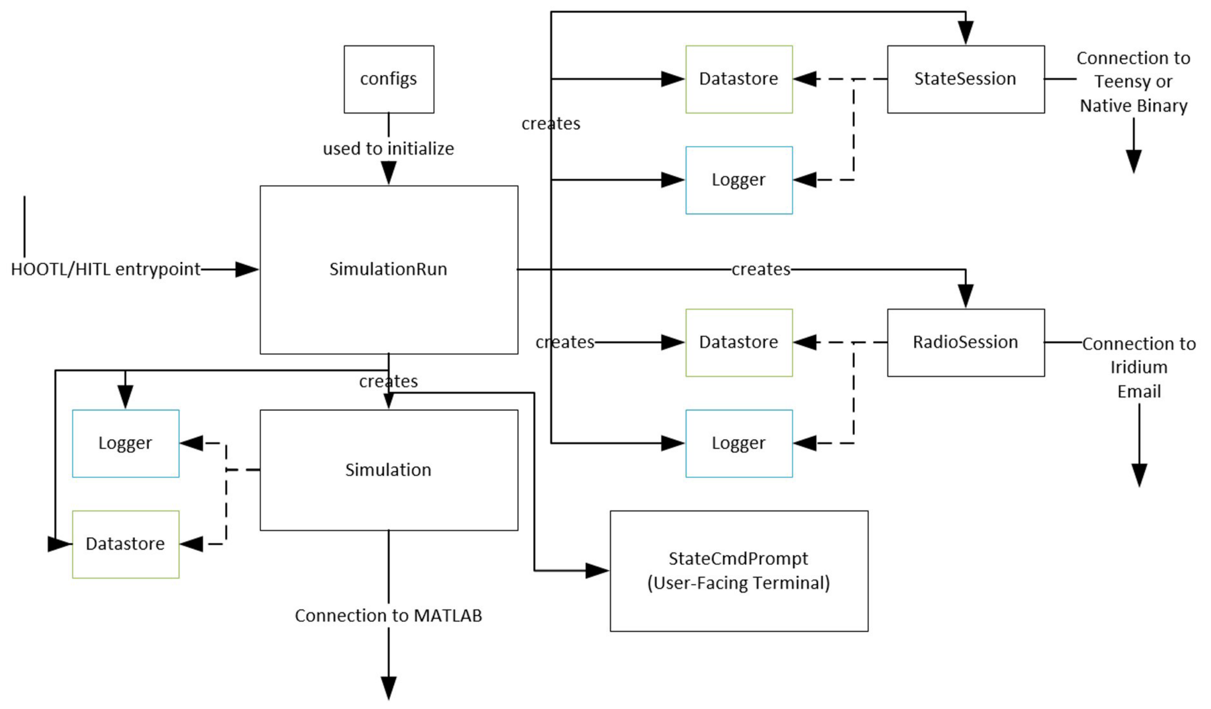
- Teensy-in-the-Loop (TITL) tests, which are like HOOTL tests but with flight software running on an actual Teensy 3.6, like it would on the satellite, rather than as a binary executable on a computer.
- Vehicle-in-the-Loop (VITL) tests, which run the flight software on the satellite's entire electronics stack. Optionally, VITL tests can incorporate other satellite elements like the propulsion system, the ADCS box, or the radio.

Flight software must be proven on HOOTL and TITL levels before running at a VITL level. This testing architecture allows us to have iterated stages of proving the flight readiness of software whilst minimizing risk to expensive hardware during testing.

See below for design documentation for ptest. To install and run ptest, consult the README in `FlightSoftware/ptest/README.md`.

PTest Software Stack

At the core of the simulation architecture is the **USBSession**. This is an integration class that either connects to a Teensy running flight software or to a desktop binary executable, and allows exchanging state fields with the flight software.



Related to the **USBSession** is the **RadioSession**, which creates a connection to the email account that PAN uses to talk to the Iridium satellite network. Using this email account, the **RadioSession** is able to interpret downlinks and send uplinks when requested by the simulation.

At the top level, the **SimulationRun** architects the **Simulation**, any **USBSession** objects, any **RadioSession** objects, the **StateCmdPrompt**, and **Datastore** and **Logger** objects required by the state sessions, simulation, and radio session.

The State Command Prompt

This is a user-facing CLI that interacts with state session and simulation objects to produce meaningful test behavior. The user has access to a wide selection of commands:

- `rs` and `ws` are used to **read state** fields and **write state** fields to a currently selected state session or radio session. `wms` can be used to write multiple state fields at a time.

  Typical usage is as follows:

```
> rs pan.cycle_no
1                                                (Completed in 604 us)
> ws cycle.start true
Succeeded                                        (Completed in 731 us)
> wms pan.state 11 cycle.start true
Succeeded                                        (Completed in 985 us)
> rs pan.cycle_no
3                                                (Completed in 651 us)
```

  You can find the actual implementation for these commands in `USBSession` and `RadioSession`.

- `cycle` is shorthand for `ws cycle.start` (advancing the flight software cycle by 1) and `cyclecount` is shorthand for `rs pan.cycle_no`.

- `plot [field]` can be used to plot the values of a state field that have been collected so far.
- `listcomp` lists the set of available state and radio sessions to connect to.
- `switchcomp` can be used to switch between state and radio sessions.
- `checkcomp` lists the currently active state or radio session. Typical state/radio session choices include
  - `FlightController`
  - `FlightControllerRadio`
  - `FlightControllerLeader`
  - `FlightControllerLeaderRadio`
  - `FlightControllerFollower`
  - `FlightControllerFollowerRadio`
- `checksim` can be used to check how many seconds are left in the current simulation.
- `endsim` ends the MATLAB simulation (though this behavior has been a little flaky.)
- `telem` is used to extract the most recent telemetry packet off of the spacecraft.
- `parsetelem` parses the most recently received packets into a meaningful state.
- `os [field]` can be used to override the value of a state field on the current state session so that values sent by the simulation for that state field value are ignored.
- `ro [field]` releases the override on a field.

Useful Commands:

- `ws cycle.auto true` : *DebugTask* in flight software will no longer wait for `cycle.start` to be true before finishing, so Flight Software cycles will automatically proceed.

PTest Cases

Take a read at *PTest Software Stack* before looking at this page. Once you do, you know that USBSession is the core of how a user interacts with flight software instances. The state command prompt provides a manual way to read and write state from flight software; ptest cases provide a powerful, Python-based, automated way to transact state fields. This allows for the creation of automated simulations and testcases on our spacecraft.

See below for an inheritance diagram of the ptest case base classes:

Writing a PTest Case

Is as simple as inheriting from either `SingleSatOnlyCase` or `MissionCase`, as diagrammed above. These base classes contain some utilities for reading and writing state to either 1 or 2 satellites, respectively.

The base ptest class also exposees a set of *FSWEnum* objects which create dual-indexing of common flight software enums (like mission state, ADCS state, etc.) by both name and numerical value. See the example below of how you can set the satellite mission state to “manual”.

Examples of writing a state field through a ptest case derived from `SingleSatOnlyCase`:

```
self.ws("pan.state", self.mission_states.get_by_name("manual"))
self.ws("dcdc.ADCSMotor_cmd", True)
self.ws("adcs_cmd.rwa_speed_cmd", [0,0,0])
```

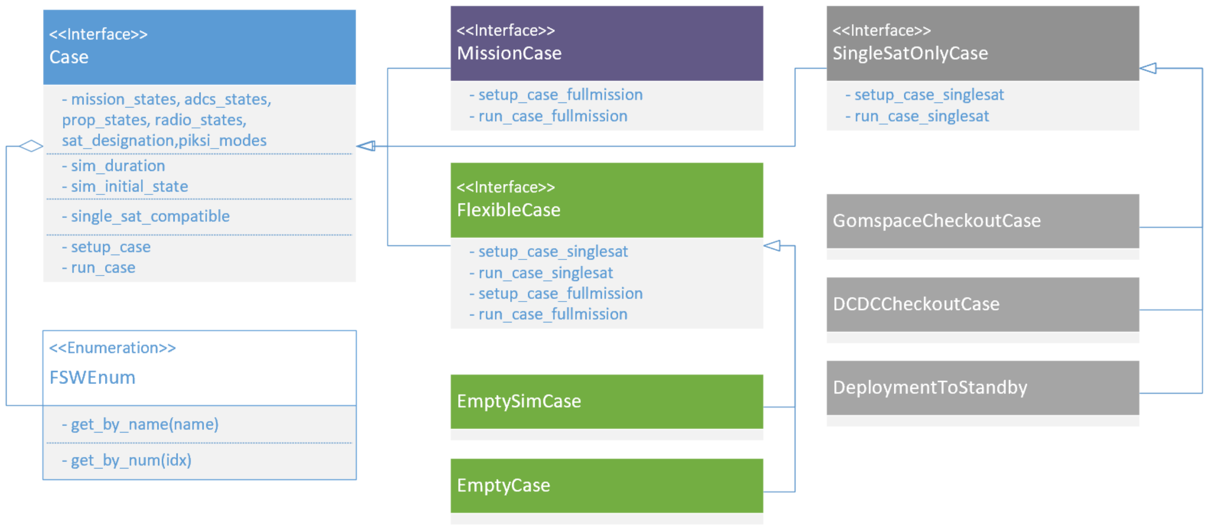


Fig. 2: Diagram depicting the relationship between the base classes of ptest cases.

`self.ws()` accepts the statefield name and a int, float, bool, or a list of them.

Examples of a reading state field through a ptest case derived from `SingleSatOnlyCase`:

```
self.rs("adcs_monitor.mag_vec")
self.rs("adcs_cmd.havt_reset0")
```

`self.rs()` returns the proper type of variable associated with each state field.
`self.rs("adcs_cmd.rwa_speed_cmd")` returns a list of floats.

Listing of Ptest Cases

ADCSCheckoutCase

The ADCSCheckoutCase

Initialization

1. Sets the mission state to `manual`
2. Sets the ADCS state to `point_manual`
3. Set the RWA mode to `RWA_SPEED_CTRL`
4. Set the intial RWA speed command to `[0,0,0]`
5. Turn on the ADCS Motor DCDC.

HAVT Checkout

The HAVT checkout begins by resetting all devices on the ADCSC.

It then pattern matches the read HAVT table with existing cataloged test-beds. If no match is found, the user is alerted.

Then all devices are disabled and reset, to make sure the devices are all properly cycled, and returned to the initially read HAVT table.

MAG Checkout

The checkout case pulls ten readings from the mag, `cycle()` ing the FC between each reading. It checks to make sure the readings change over time, and that the magnitude of the readings are reasonable (within expected earth magnetic field strength expectations).

MAG Independence Checkout

If both magnetometers are functional, this test section will disable MAG1, and check that MAG2 still works. The same checkout is performed on MAG1 with MAG2 disabled. The independence checkouts re-run the same checkouts as above.

GYR Checkout

The checkout case pulls ten readings from the gyro, `cycle()` ing the FC between each reading. It checks to make sure the readings change over time, and that the magnitude of the readings are reasonable.

Wheel Checkout

The wheels are put into `RWA_SPEED_CTRL` and it checks that the speed read is reasonably close to the speed read for a series of tests.

The wheels are then put `RWA_TORQUE_CTRL` and it checks that the torque read is close to the actual torque read for a series of tests.

SSA Checkout

Each Sun Sensor is polled for its voltage, and the `sum_of_differentials` method is used to check that the sensors change over time. Alternatively it is just manually inspected.

TODO FURTHER CHECKOUTS

PiksiCheckoutCase

The PiksiCheckoutCase continuously reads data from the Piksi device.

It first performs a series of 10 readings for the user to glimpse at the data coming in. Then a series of 100 readings are performed. It checks to make sure that the most common mode of return is a nominal one or a nominal combination.

If not a TestCaseAssertion is thrown, indicating that the Piksi is likely not functioning as intended.

For each vector that is available, it that the vector changes with time, and that the magnitude of the entire vector is within expected bounds.

Position should be within 10 km of 6371 km. Velocity is within 4 m/s, and that the baseline vector is within 100 m.

The user must then check that the Piksi is functioning as expected from the diagnostic data given the condition of the test bed.

Deployment to Initialization Hold Checkout Case

This checkout case confirms that, if the satellite is deployed and all the ADCS devices are functional, then the satellite will move to `detumble`. If the satellite is deployed and if one or more of the ADCS devices are not functional, then the satellite moves to `initialization hold`.

First, the checkout case moves the satellite to `startup` and waits the full deployment period. Then, the checkout case tests each of the following scenarios:

1) **All ADCS Devices are functional:** The case unsignals all the listed faults and checks that the satellite moves to `detumble`.
2) **ADCS is not functional:** This fault could occur if I2C communication between the flight computer and the ADCS fails. The checkout case will signal the `adcs.functional` fault and check that the satellite moves to `initialization_hold`.
3) **ADCS Wheels 1-3 are not functional:** The ADCS wheels allow the satellite to adjust its orientation in space. The case will trip each of the ADCS wheel faults, one-by-one, and check that the satellite moves to `initialization_hold` each time.
4) **ADCS Potentiometer is not functional:** The ADCS potentiometer is a variable resistor that controls the torques that the motors operate with. The checkout signals `adcs_monitor.wheel_pot_fault` and checks that the satellite moves to `initialization_hold`.

DCDC Checkout Case

The DCDC checkout case verifies that we are able to control the values of the DCDC pins: `ADCSMotorDCDC_EN`, which brings power to the ADCS system, and `SpikeDockDCDC_EN` which brings power to the propulsion and docking systems. The checkout case also verifies that we can reset and disable these pins from the ground.

GomspaceCheckoutCase

The GomspaceCheckoutCase tests the readable state fields read from the Gomspace and compares them to the expected values from its datasheet and manual. The case also tests that writable fields can be properly set and that powercycling is successful. The test case will log any incorrect or unsuccessful reads/writes.

DockingCheckoutCase

The DockingCheckoutCase is meant that we can write and read to the docking system's state fields, and in HITL is also meant to test that the motor turns the expected amount with the currently set values for step angle and delay.

The test case logs the state of all the fields as it runs. It first checks that the configuration is undocked in the undocked magnet configuration and not turning, and that the configuration is commanded to be in the docked position. The initial step angle and delay are verified, and then the system is sent the command to undock and then to dock again. Then, the step angle and step delay are written to different values and the process is repeated. There should be a noticeable difference in speed, but overall each dock and undock command should take around a minute.

CHAPTER

TWO

GNC AND SIMULATION SOFTWARE DOCUMENTATION

The repository for GNC and simulation software is here.

2.1 Introduction

There are two main deliverables contained within the PSim repository: the `gnc` library and the `psim` Python module. Here we'll provide a brief overview of the motivation behind these two products, their respective build systems, and their general layout within the repository.

2.1.1 GNC Library Overview

The `gnc` Library is an upstream dependency of PAN's flight software and provides it with implementations of estimators, controllers, and environmental models – among other things.

With microcontrollers being the eventual target platform, the entire `gnc` library is written in C/C++, avoids dynamic memory allocation, and is built using the PlatformIO build system. The general layout of the `gnc` library within the repository is outlined below:

- The header files made public by the library are located under the include/gnc and include/orb directories.
- The source files compiled as part of the library and header files that are considered private to the library's implementation are located under the src/gnc directory.
- Upstream dependencies are pulled in as git submodules and are located within the lib directory.
- Unit tests for the library are located in test/gnc.
- Build target information is enumerated in platformio.ini and the file specifying the `gnc` library to PlatformIO is the library.json file.

For more details on building the `gnc` library and running tests please see *Building and Testing GNC*.

2.1.2 PSim Overview

The `psim` Python module is intended to serve two main purposes:

- Support high-fidelity integrated testing both in HOOTL and HITL configurations. This means `psim` must be capable of simulating orbital and attitude dynamics, realistically respond to the actuator commands issued by flight software, and respond with simulated sensor data that can be fed back into flight software.
- Serve as a simulation environment to verify estimator and controller implementations contained with the `gnc` library independant of flight software. The main reason to verify `gnc` algorithms outside the integrated testing

environment described above is speed. The above tests only run at real time meaning verifying an orbit estimator, for example, over the course of many orbits in such an environment would be a very inefficient process.

The first use case is supported by the rudimentary interface provided by the `psim.Simulation` object. It allows `ptest` to step the simulation as required and easily exchange sensor and actuator data as needed. The second use case is satisfied by the `psim` command line interface described in *Running a Simulation*. It allows the user to run a "standalone" simulation running far faster than realtime, manipulate initial conditions, generate plots, et cetera.

Given `psim` only needs to be compiled for desktop platforms we have more freedom to use the full set of C/C++ and STL features. The Bazel build system is also use making handling a large number of dependencies, large number of build targets, and autogenerated source files far easier to deal with. The general layout of `psim` within the repository is given below:

- Additional Skylark to adapt the Bazel build system for use with `psim` is located within the bazel directory. The code here supports the autogenerated **.yml.hpp* files among other features.
- The public header and YAML model interface files for each `psim` library are located within the include/psim directory.
- The source files compiled are part of each respective library and header files considered to be private to each library's implementation are located under the src/psim directory.
- Unit tests written using GoogleTest are locating in test/psim.
- The source code that actually makes the `psim` code accessible via Python is defined in python/psim/_psim.cpp.
- C/C++ build targets and dependency information are defined in the WORKSPACE and BUILD.bazel files.
- Additional Python code making up the `psim` Python module can be found under the python directory.

For more information on building the `psim` Python module and running unit tests please see *Building and Testing PSim*.

2.2 Installation Guide

This outlines how to setup your development environment for the `psim` repository. If you're experiencing build issues, try searching through *Common Problems* and seeing if anything seems relevant to you. Feel free to expand on that section if necessary as well!

2.2.1 Dependencies

All of the following must be installed on your system in order to build the `gnc` and/or `psim` libraries:

- Python three (Python 3.6 or higher).
- Python three development headers.
- Python three `distutils` library.
- Bazel build system.

See this Bazel installation guide for more details on setting up Bazel.

Python development headers can frequently be installed via your systems package manager. On a Debian system, as an example, use:

```
sudo apt-get install python3-dev
```

The Python `distutils` package always seems to already be installed. If unsure, continue with the installation guide and revisit the `disutils` package later if needed.

2.2.2 Cloning the Repository

Be sure the clone the repository recursively to download all submodules as well:

```
git clone --recursive git@github.com:pathfinder-for-autonomous-navigation/psim.git
cd psim
```

When pulling in updates or switching between branches with different submodule commits, you must run:

```
git submodule update
```

to actually reflect changes inside the submodules themselves!

2.2.3 Python Virtual Environment

Warning: If you do not use your system's default version of Python three, `psim` may fail to build or run.

Both `gnc` and `psim` will share a Python virtual environment development. From the root of the repository, run the following:

```
python -m venv venv
source venv/bin/activate
pip install --upgrade pip wheel
```

being sure to use you're system wide install of Python three.

2.2.4 Building and Testing GNC

The `gnc` library is build and tested using PlatformIO. From within the *Python Virtual Environment*, execute the following commands to install the required dependencies:

```
pip install -r requirements.txt
```

and then you are free to build and run the `gnc` unit tests natively with:

```
pio test -e native
```

There are other build targets to run CI, execute code on a Teensy microcontroller, etc. I recommend checking out the repositories platformio.ini for more information on the various build targets.

Further testing of the `gnc` library is possible from within `psim` itself – more on this later.

2.2.5 Building and Testing PSim

The first way to interacting with `psim` is by building and running it's suite of C++ unit tests. This is done by executing the following:

```
bazel test //test/psim:all
```

A more limited set of unit tests that are executed for CI can also be run with:

```
bazel test //test/psim:ci
```

The second, and far more useful, way of using `psim` software is building the `psim` Python module. Prior to doing so, however, you must install the `lin` Python module in your *Python Virtual Environment* with:

```
pip install lib/lin
```

This should be reinstalled everytime the `lin` submodule receives updates – this isn't too often nowadays. From there, the `psim` module is installed via:

```
pip install -e .
```

where the `-e` flag installs the Python package in "editable mode" and allows Bazel build caching system to greatly reduce build times – because a new copy of the repository isn't created for each install.

To verify the `psim` module is installed and functioning, run:

```
python -m psim --help
```

and, if interested, continue on to *Running a Simulation* to run a full simulation with your new Python module.

2.2.6 Intellisense Support

Bazel doesn't play nicely with VSCode's existing C++ intellisense extensions on it's own. In order to get intellisense working with the Bazel targets, run the following setup command once:

```
./tools/bazel-compilation-database.sh
```

which will install a scripting tool called `bazel-compdb` to `~/.local/bin` on your machine. Please ensure you have a `~/.local/bin` directory on your machine and it's in your `PATH`.

Note: It's possible to install the tool in a different directory if you'd prefer not to use `~/.local/bin`. Simply edit the `tools/bazel-compilation-database.sh` script accordingly.

Once installed, simply call:

```
bazel-compdb
```

periodically from the root of the repository. It will generate a `compile-commands.json` file that the VSCode extensions will read and intellisense should start working.

2.2.7 Common Problems

Bazel Requiring Python Two

In the past, we've seen Bazel trying to determine the version of a system wide Python two installation. It will error out and complain that a command similar to:

```
python --version
```

failed to run. There are two ways we're currently aware of to fix this:

- Alias/install the Python three installation as the default Python on your system. Arch linux and other operating systems do this by default and `psim` builds without a Python two installation.
- Install Python two on your system even if you aren't going to use it. Bazel will be smart and figure out `python3` still exists on your system and use that Python version instead.

Bazel Failing to Build PSim After Upgrading Python

Installing `psim` with:

```
pip install -e .
```

will fail after performing a major version upgrade of Python three on your system – e.g. upgrading Python 3.8.x to Python 3.9.x.

In the build process, Bazel hunts down the include path to your current Python three development headers. It then creates a symbolic link to that directory which is passed as an include path to the compiler at build time. That symbolic link becomes invalid when upgrading through a major version of Python because the include directory name changes. As such, Bazel will spit out compiler errors saying things like the header `Python.h` can't be found.

To fix this you should run the following in the root of the repository:

```
bazel clean --expunge
```

and then, if you haven't already create a new virtual environment and repeat the install process. Running `bazel clean --expunge` forces Bazel to once again hunt down the Python include path fixing the issue.

PSim Standalone has Issues Generating Plots

This has been noticed to happen on MacOS a couple times. Recreating you're *Python Virtual Environment* with the `--system-site-packages` flag may help:

```
rm -r venv
python -m venv venv --system-site-packages
source venv/bin/activate
...
```

The Nuclear Option

If all else fails, it's worth trying to clone a fresh copy of the repository and attempt setup again from the beginning. It has been necessary on a couple rare occasions where we've never been able to reproduce the error.

2.3 Running a Simulation

This gives an overview of how to run a PSim standalone simulation through via the command line. If you're looking for details on the Python classes `psim.SimulationRunner` or `psim.Simulation` please check out python/psim/simulation.py.

Prior to running a simulation you must have installed the `psim` Python module as described in *Building and Testing PSim*.

2.3.1 Command Line Interface

Note: Only the core options provided by the command line interface are presented here. For a complete list please run `python -m psim --help`.

The core usage of the `psim` is given by:

```
python -m psim [-p PLOTS] [-ps PLOTS_STEP] [-s STEPS] -c CONFIGS SIM
```

where:

- `-p PLOTS`, `--plots PLOTS` specifies a comma separated list of plotting configuration files used to determine what data to log and plot over the course of the simulation.

  These plotting files can be found within config/plots and it's nested subdirectories. To refer to a particular plotting file you specify it's relative path from `config/plots` dropping the `.yml` suffix.

- `-ps PLOTS_STEP`, `--plots-step` instructs the simulation how frequently it should poll data from the simulation for plotting in simulation steps.

  For example, setting this to one provides a plot data point at every step while setting it to ten would only log a data point from plotting every ten steps.

- `-s STEPS`, `--steps STEPS` specifies how many steps the simulation should run for (a value of zero allows the simulation to run forever).

  There are currently no other stopping conditions provided by the command line interface.

- `-c CONFIGS`, `--configs CONFIGS` specifies a comma separated list of configuration files specifying initial conditions.

  These configuration files can be found within config/parameters and it's nested subdirectories. To refer to a particular configuration file you specify it's relative path from `config/parameters` dropping the `.txt` suffix.

- `SIM` gives the simulation type to be run.

  The string name of the Python simulation type is passed here and is case sensitive. A list of all the available types can be found in python/psim/sims.py.

2.3.2 Testing the Attitude Estimator

As an example, how to run a test of the attitude estimator is given below:

```
python -m psim -s 2000 -p fc/attitude,sensors/gyroscope,truth/attitude -ps 1 -c␣
↪sensors/base,truth/base,truth/deployment AttitudeEstimatorTestGnc
```

This runs a simulated test of the attitude estimator starting during deployment. Once the simulation terminates, you should be left with tons of plots describing the performance of the attitude estimator, the gyroscope, and the truth attitude dynamics.

Feel free to mess around with different simulation types, various initial conditions, and/or other plotting configurations. Do note, however, that all combinations don't always produce a "valid" simulation. For example, you can't ask for `truth/attitude` plots when running a simulation without attitude dynamics - e.g. `SingleOrbitGnc`.

2.4 PSim Architecture

2.4.1 Introduction

2.4.2 Configurations

Configuration Files

2.4.3 State Fields

All data sharing between *models* within a *simulation* as well as transactions to and from Python happen via state field reads and writes. Here, the two main types of supported state fields will be described along with their intended use cases.

Lazy State Field

Lazy state fields, as suggested by their name, are lazily evaluated when accessed given the current state of the simulation - i.e. the value of other state fields. The result of a lazy evaluation is cached for the remainder of the simulation step to allow for low-overhead duplicate accesses and reset when the simulation is stepped forward again.

The main motivation for supporting lazily evaluated fields is two fold:

- Improved performance. The end user shouldn't pay for the computation of fields that aren't strictly required by their use case. Most lazy fields implement convenience coordinate transformations, calculate sensor error calculations, estimator performance metrics, et cetera that aren't required by most use cases.
- Convenience. While not paying to computational overhead that you don't need is nice, having the ability to at anytime query the value of some lazily evaluated field for debugging purposes is invaluable. Furthermore, it reduces potential code duplication for particular use cases if the simulation can already provide a large swath of information via lazy evaluation.

The implementation of a lazy state fields can be found in include/psim/core/state_field_lazy.hpp. Please refer the documentation describing *models* for more information on how lazy state fields are implemented and used with a simulation.

Valued State Field

A valued state field is, again as the name suggests, backed directly by a value in memory. There is no lazy evaluation and a valued field can be written too if marked as writable by the owning model.

Generally speaking, a valued state field is used to store fields integral to the current state of the spacecraft. Something like each satellite's position and velocity can't really be lazy evaluated; on each simulation step the dynamics model must propagate the position and velocity forward in time. Valued fields can also be used if the data is calculated on each step anyway even if it's not "integral" to the state of the simulation. An example of this would be the gyroscope bias estimate determined by the attitude filter.

include/psim/core/state_field_valued.hpp contains the implementation of valued state fields. Again, please refer the documentation describing *models* for more information on how valued state fields are implemented and used with a simulation.

2.4.4 Models

Model

Model Interfaces

Model List

2.4.5 Simulations

2.4.6 Python

CHAPTER

THREE

GROUND SOFTWARE DOCUMENTATION

This section of doumentation will contain information about ground software. Ground software has two purposes:

1) Store and index telemetry coming from the satellites
2) Send commands to the satellite via the Iridium Satellite Constellation Network.
3) Send commands to the Flight Computer to test telemetry without hardware in the loop.

The code for the telemetry parsers and MCT is available here.

The code for the downlink processing server and telemetry software is available here.

3.1 Radio Session

The RadioSession class represents a connection session with a Flight Computer's Quake radio. RadioSession is used by the simulation software and the user command prompt to read and write to a flight computer's Quake radio. Upon startup, a radio session will create HTTP endpoints which can be used to send telemetry to a radio. This section elaborates on these endpoints, how autonomous uplinks are scheduled, and the two main methods in radio session: read_state and write_state.

3.1.1 Uplink Timer

The two satellites communicate their respective GPS positions and other state information via the ground station and Iridium. After recieving state information from one satellite, the ground autonomously sends an uplink with the relevant information to the other satellite. While communication between the satellites and the ground station is established, we can expect information packets to be recieved and sent by the ground every few minutes.

After autonomously creating a packet to be sent to a satellite, the radio session queues the packet and starts an `Uplink Timer`. The radio session waits until the timer is up before sending the uplink to the other satellite. The amount of time that the radio session waits before sending the uplink can be configured in the radio session config file. In the config file, the `send_queue_duration` specifies the total amount of time that radio session waits before sending the uplink and the `send_lockout_duration` specifies the amount of time during which the mission commander can no longer make edits to the queued uplink. For example, if the `send_queue_duration` is 10 minutes and the `send_lockout_duration` is 2 minutes, then the mission commander can make edits to the queued uplink for only the first 8 minutes. After 10 minutes, the uplink will be sent to the satellite via Iridium.

The `Uplink Timer` can be paused and resumed to allow more time for the mission commander to edit a queued uplink. This can be done by sending a request to a designated HTTP endpoint.

3.1.2 HTTP Endpoints

Upon creation, the radio session will set up four HTTP endpoints that allow the mission commander to send uplinks to the satellite. The mission commander can access these endpoints and send telemetry using NASA's OpenMCT interface.

1) Time

This endpoint returns the amount of time left on the uplink timer is an autonomous uplink is queued.

2) Pause

This endpoint allows the mission commander to pause the uplink timer so that he or she can make edits to a queued uplink.

3) Resume

This endpoint allows the mission commander to resume the uplink timer once he or she is done making edits to a queued uplink.

4) Request Telemetry

This endpoint allows a mission commander to send telemetry to a satellite by posting requested telemetry as a JSON object over HTTP. If an autonomous uplink is queued to be sent, then the requested telemetry will be added to the queued uplink packet. We are constrained to send 70 bytes of information per uplink packet. Therefore, editing queued autonomous uplinks allows us to send as much information per uplink packet as possible.

On the other hand, if there is no autonomous uplink queued, then the uplink packet will immediately be sent to the satellite (i.e there will be no use of an `UplinkTimer` or a queue duration).

3.1.3 Read State

read_state() allows us to read the most recent value of a statefield of a satellite. To do this, RadioSession establishes a connection to the Email Processor responsible for indexing statefield information from both satellite radios. RadioSession then sends a GET request to the email processor over HTTP with the name of the ElasticSearch index (statefield_report_[imei of RadioSession's connected radio]) and the name of the desired statefield as queries.

3.1.4 Write State

write_state() allows us to set the value of a statefield for a specific satellite from the ground. First, RadioSession confirms whether or not there are any uplinks currently queued to be sent to the connected radio. RadioSession does this by sending a GET request to the ElasticSearch database over HTTP with the name of the ElasticSearch index (iridium_report_[imei of RadioSession's connected radio]) and the name of the *send-uplinks* flag as queries.

If RadioSession is cleared to send uplinks, then RadioSession 1) waits a designated send queue duration and then 2) sends an uplink to the satellite via Iridium. The subject of the uplink is the IMEI number of the radio of the satellite that will set the statefields. The uplink message will also contain an attached SBD file that holds the serialized names and the desired values of the statefields we wish to set from the ground. While the uplink is queued, the mission commander can edit the contents of this uplink packet over HTTP.

Once the radio recieves the uplink, the Uplink Consumer in Flight Software will then deserialize and read the SBD file and set the statefield values accordingly.

3.1.5 Email Access

All uplinks are sent to the satellites via the Iridium Satellite Network. To accomplish this, radio session sends an email from the designated PAN email account to Iridium's email address. The subject of the email is the IMEI number of the satellite's radio, and attached to the email is a file holding a serialized uplink packet.

In order to access PAN's email account from a remote server and send messages, the radio session obtains token credentials using Google's Gmail API. We are constrained to a token grant rate limit a 10,000 grants per day, or approximately 10 token grants a minute. However, since the radio session waits an established send queue duration before sending an uplink, this limit poses no foreseeable issue.

3.2 State Session

A state session represents a connection session with a Flight Computer's state system. It is used by the simulation software and user command prompt to read and write to a flight computer's state. State sessions can be used for testing telemetry without hardware in the loop.

Upon startup, a state session will create two HTTP endpoints which can be used to send telemetry to the flight computer. This section elaborates on these endpoints and how to send telemetry packets.

3.2.1 Sending Telemetry

We can request to send telemetry to the flight computer by running `uplink [field1] [val1] [field2] [val2] ...` in the user command prompt. The state session will then collect the requested telemetry as a JSON object and serialize this information using the Uplink Producer.

A state session can then send the serialized uplink packet to the flight computer by sending a JSON command with the uplink packet to the flight computer's debug console. The debug console will parse the uplink packet and move the packet into the Quake Manager's radio MT buffer. The Quake Manager would then deserialize and read the uplink packet on the next control cycle.

3.2.2 HTTP Endpoints

Upon creation, the state session will create a single http endpoint that allows the commander to request to send telemetry to the flight computer.

3.3 Processing and Storing Downlinks

3.3.1 ElasticSearch

ElasticSearch is a database that allows us to organize information into various indexes or subcategories. In Ground software, every radio has two indexes in which we store telemetry data as JSON objects:

1. Iridium Reports
    - Name: Iridium_Report_[IMEI of the radio]
    - Iridium reports house the most recent information about telemetry, such as Mobile Originated Message Numbers (MOMSN), Mobile Terminated Message Numbers (MTMSN), and confirmation MTMSNs. An MOMSN number is the ID of the most recent downlink recieved. An MTMSN number is the ID of the most recent uplink sent from the ground. A confirmation MTMSN is what we call the MTMSN of the last uplink message recieved and processed by the radio. If the most recent confirmation MTMSN and the most recent MTMSN are not equal, then that means that there is a message queued in Iridium that has not yet been recieved by the satellite radio, and we prohibit the ground from sending any more uplinks by setting the *send-uplinks* flag in the Iridium report to False.

2. Statefield Reports (Statefield_Reports_[IMEI of the radio]).
    - Name: Iridium_Report_[IMEI of the radio]
    - Statefield Reports house the most recent information pertaining to the actual satellite(s). This information is found in the Short Burst Data (SBD) attachment in downlink emails.

Every radio has their own Iridium Report index and Statefield Report index to store telemetry information in ElasticSearch. This allows us to distinguish the statefield and telemetry information for each satellite.

3.3.2 Email Processor

Telemetry is sent from the satellite to the PAN email account via the Iridium Satellite Constellation Network in compressed serialized packets. These packets contain special information and data about the satellites that we need to store and index. This is accomplished by a server written with Flask which continuously reads unread emails from the Iridium Network, parses the data that comes from the satellite in the form of an email attachment, and stores that parsed information in an Elasticsearch database.

When the email processor is started, it opens a thread *check_email_thread*, which will continuously do the following:

1. Read the most recent unread email received from the Iridium email account.
    - If the most recent unread email is identified as a downlink from a satellite radio, the server parses the information stored in the email attachment containing statefield data and returns a statefield report. A statefield report is a JSON object that holds statefield names, the updated values of each statefield, and the time at which the report was recieved.
    - If the most recent unread email is identified as a confirmation that a radio has received an uplink, the server will record that an uplink confirmation was just received and return None.
    - If there are no unread emails from any satellites, the server returns None.
2. Process the information recieved from the most recent unread email from Iridium
    - If the server has recieved a statefield report, then the thread indexes the statefield report in ElasticSearch. The function will also create and index an Iridium report.
    - If the server does not recieve a statefield report, but we see that the class variable for the uplink confirmation is set to true, then the thread creates and indexes only an Iridium report in ElasticSearch.
    - If the server has not recieved any sort of communication from the satellite, then we do nothing.
3. The thread delays for 10 seconds to reduce CPU bandwidth

3.3.3 Reading Stored Telemetry

The email processor also has an endpoint from which we can access data from the ElasticSearch database. This endpoint requires two queries: the IMEI number of the radio you want telemetry information from, and the specific statefield that you want to know the most recent value of. The Flask server will then search for the statefield report index based on the given IMEI number, and the search within that index for the value of the most recent statefield that was requested.

This endpoint is used for reading statefield information from a satellite when opening a RadioSession to a certain radio. It is also used by RadioSession to confirm whether or not the ground is cleared to send more uplinks (i.e if there aren't any messages already queued to be sent to the satellite).

3.4 Telemetry Management

The data coming from the satellite must be serialized and compressed because we are only able to send 70 bytes of information at a time over radio. Thus, it is necessary to compress and then parse data from the satellite to ensure we can recieve as much information as possible whenever we are able to establish communication.

I recommend reading these two sections to better understand how downlink information is compressed and sent over radio to get a better sense of how downlink data is parsed:

- How satellite information is serialized.
- How serialized satellite information is organized and sent over radio.

3.4.1 Uplink Producer

The uplink producer accepts a JSON file containing the names and desired values of the statefields to be set from the ground. The producer then serializes all the statefield information to a bitstream and writes the uplink packet to an SBD file. The uplink producer throws an error if the size of the serialized requested telemetry exceeds the limit of 70 bytes.

3.4.2 Downlink Parser

As the thread in the email processor reads unread emails from the Iridium network, the Downlink Parser parses the serialized information and data into a readable JSON object (ElasticSearch only accepts JSON objects).

The downlink parser reads files/packets containing the statefield information of a groups of flows with varying priorities and processes them at a bit level. If the first bit of a packet is 1, then that signifies the start of a new downlink frame. The downlink producer will continue to read serialized data until it recieves another frame that starts with 1. Once the downlink parser reads the next packet that starts with 1, that means the previous frame is finished and the downlink parser returns the most recently collected frame as a JSON object.

CHAPTER FOUR

FIRST TIME SETUP:

Make sure you have bazel installed Make sure you have your venv installed and ready to going Make sure you have elasticsearch instaled: This link worked for me on Ubuntu 20.04 (WSL2): https://www.digitalocean.com/community/tutorials/how-to-install-and-configure-elasticsearch-on-ubuntu-20-04

CHAPTER

FIVE

EVERY TIME SETUP:

From the FSW repo: Make sure your submodules are up to date:

```
git submodule update --init --recursive
```

Make sure that PSim (within FSW):

```
pio run -e lib/common/psim
```

Make sure that the builds of fsw that you're going to be using are built: For example:

```
pio run -e fsw_native_leader
pio run -e fsw_native_follower
pio run -e fsw_native_leader_autotelem
```

CHAPTER

SIX

SPOOLING UP THE STACK:

Start ElasticSearch: If you don't have systemd:

```
sudo -i service elasticsearch start
```

On MacOS:

```
elasticsearch
```

If you do have systemd, you can set elasticsearch to startup everytime with: https://www.elastic.co/guide/en/elasticsearch/reference/current/starting-elasticsearch.html

In a seperate terminal: Start TLM:

```
python -m tlm
```

In a seperate terminal: Start an AutonomousMissionController with:

```
python -m ptest runsim -c ptest/configs/amc.json -t AutonomousMissionController
```

In a seperate terminal: Start MCT inside FlightSoftware/MCT: Make sure to specify a ptest/config that uses a specific config:

```
cd MCT
npm start ptest/configs/hootl_hootl_autotelem.json
```

6.1 Installation Guide

This outlines how to setup your development environment to run full mission simulations. If you encounter any issues during installation, please first refer to the *Common Problems* page. Feel free to add an additional section to the *Common Problems* page if you encounter a new issue.

6.1.1 Dependencies

In order to run full mission simulations, the dependencies for flight software, simulation software, and the ground software stack must be installed. Flight and simulation software dependencies are covered in detail in the PSim *Dependencies* guide – flight software's dependencies are a subset of PSim's.

In addition to the above dependencies, we also need to install ElasticSearch and the NodeJS package manager NPM for OpenMCT support. Installation for these packages is system specific but generally can be installed via your OS' package manager.

If you're running on WSL this may be helpful for ElasticSearch.

6.1.2 Installing Flight Software

Installing the flight software repository is the only code-base required for full mission simulations. Simply clone the repository recursively

```
git clone --recursive git@github.com:pathfinder-for-autonomous-navigation/
↪FlightSoftware.git
cd FlightSoftware
```

and setup a virtual environment:

```
python -m venv venv
source venv/bin/activate
pip install --upgrade pip wheel
pip install -r requirements.txt
pip install -e lib/common/psim
```

being sure to use your system's default version of Python 3 to create the virtual environment as required by PSim in *Python Virtual Environment*. Assuming all has gone well so far, both the `psim` and `ptest` Python modules have been successfully installed. For help with `psim` specific install instructions, please see PSim's *Common Problems* section.

6.1.3 Installing Ground Software

For detailed instructions please reference the MCT README.

6.2 Running a HOOTL HOOTL

This outlines how to start a HOOTL-HOOTL, full-mission testcase with ground software in the loop. Thanks to the generally abstract interface provided by `ptest` this looks very similar to *Running a HOOTL HITL*.

In general, it's recommended to start all of processes described here in separate terminals for ease of use.

6.2.1 Starting ElasticSearch

First on the list here is to start ElasticSearch on your machine. On Linux systems with `systemd` this is generally done with

```
sudo systemctl start elasticsearch.service
```

and stopped with

```
sudo systemctl stop elasticsearch.service
```

I'd recommend checking out this Arch Wiki page for information on basic `systemd` usage for more commonly used commands.

For those running on Mac, ElasticSearch can be started with

```
elasticsearch
```

In either case, once ElasticSearch is booted it's recommended to clear the database with

```
curl -XDELETE localhost:9200/*
```

to prevent old data from interfering with the simulation.

If the above directions don't help with starting ElasticSearch, it may be worth checking out their guide here as well.

6.2.2 Starting PTest

From the root of the Flight Software repository the desired `ptest` case can be started with

```
python -m ptest runsim -c ptest/configs/hootl_hootl_autotelem.json -t␣
↪DualSat[(Startup)(Detumble)(Standby)(FarField)(NearField)]Case
```

where the testcase name boots into the desired mission scenario (either startup, detumble, standby, near field operations, or far field operations). Note that, generally speaking, the autotelem feature is desired for full mission cases so OpenMCT actually gets populated with data. This is why we're running with a `*_autotelem.json` configuration.

Please remember to configure the IMEI numbers in the `hootl_hootl_autotelem.json` file. Failing to make these numbers unique to your own machine could cause email collisions between simulations being run by different PAN members.

6.2.3 Starting the Autonomous Mission Controller

The autonomous mission controller (AMC) can be starting with

```
python -m ptest runsim -c ptest/configs/amc.json -t AutonomousMissionController
```

where it's absolutely critical to match the IMEI number within the `amc.json` configuration to those used in the testcase.

6.2.4 Starting OpenMCT

Assuming OpenMCT was already installed, the server can be started with

```
cd MCT
npm start ../ptest/configs/hootl_hitl_autotelem.json
```

6.3 Running a HOOTL HITL

This outlines how to start a HOOTL-HITL, full-mission testcase with ground software in the loop. Thanks to the generally abstract interface provided by `ptest` this looks very similar to *Running a HOOTL HOOTL*.

In general, it's recommended to start all of processes described here in separate terminals for ease of use.

6.3.1 Starting ElasticSearch

Please follower the instructions include with the HOOTL HOOTL simulation instructions for *Starting ElasticSearch*.

6.3.2 Starting TLM

Because in a HOOTL HITL setup we're going to have one actual Iridium radio, it's neccesary to start the `tlm` service. This can be done with

```
python -m tlm
```

6.3.3 Starting PTest

From the root of the Flight Software repository the desired `ptest` case can be started with

```
python -m ptest runsim -c ptest/configs/hootl_hitl_autotelem.json -t␣
↪DualSat[(Startup)(Detumble)(Standby)(FarField)(NearField)]Case
```

where the testcase name boots into the desired mission scenario (either startup, detumble, standby, near field operations, or far field operations). Note that, generally speaking, the autotelem feature is desired for full mission cases so OpenMCT actually gets populated with data. This is why we're running with a `*_autotelem.json` configuration.

Please remember to configure the IMEI number for the HOOTL instance in the `hootl_hitl_autotelem.json` file. Failing to make this number unique to your own machine could cause email collisions between simulations being run by different PAN members.

6.3.4 Starting the Autonomous Mission Controller

The autonomous mission controller (AMC) can be starting with

```
python -m ptest runsim -c ptest/configs/amc.json -t AutonomousMissionController
```

where it's absolutely critical to match the IMEI number within the `amc.json` configuration to those used in the testcase. Note that in the case of HOOTL HITL you'll need to pull the actual IMEI number from the hardware quake itself!

6.3.5 Starting OpenMCT

Assuming OpenMCT was already installed, the server can be started with

```
cd MCT
npm start ../ptest/configs/hootl_hitl_autotelem.json
```

6.4 Common Problems

6.5 Debugging

Intercept a specific value out of elasticsearch:

```
???
```

Start ElasticSearch:

```
sudo systemctl start elastic ElasticSearch
```

Look at what is inside elasticsearch: .. code:: bash

 curl http://localhost:9200/_aliases?pretty=true

Clear ElasticSearch: .. code:: bash

 curl -X DELETE "localhost:9200/_all?pretty"

If you ever see any problem with PSim first try this!!!!

```
git submodule update --init --recursive
pip install -e lib/common/psim
```

Anything with the whole MCT/PTest Stack: Are you sure elasticsearch was started? Are you sure tlm was started? Are you sure PTest is running? Are you sure MCT is running?

6.6 Starting up FAQ

Something broke please help idk fill this in

Start ElasticSearch:

```
idk lmao
```

In a seperate terminal: Start TLM:

```
python -m tlm
```

CHAPTER

SEVEN

LESSONS FROM PAN

This section of the documentation contains documentation on both the flight software and the PAN satellite's subsystem architectures. The two ideas go hand-in-hand, which is why their documentation is woven together. This section is primarily meant to be *design* documentation, although there is some user documentation as well.

7.1 Mechanical

Any power supply of a subsystem that we need to verify is "on" should have its own indicator LED light or it should have contacts on the exterior to check with a multimeter.

Every board that required a firmware upload, or is configurable, MUST have an connection available from the exterior satellite in an assembled state.

The exterior should have better grips or handles to hold the satellite from if possible.

All buttons that need to be pressed, such as reset buttons should also be moved or easily accessable.

7.2 ECE

Magnets shall not be placed near motors. This prevents PAN's docking system from working reliably.

Magnetomers shall be placed far from motors because their readings will become noisier.

As much as possible step up voltage systems are to be avoided. Instead, if the battery voltage is 8.4V tops, try to find motors that operate on less than 8.4V. Sensor boards should have no reason to operate on 24V as well.

Avoid ground loops.

Make sure grounds are shared.

The spacecraft should be able to hook up a power supply in a way that mimicks the solar panels so that software that permits charging or controls charging can be tested in a flight like manner.

Avoid long bus communication lines. These will act like antenna and put strain on signal quality.

For god sake. PLEASE PLEASE PLEASE. Avoid serial communication that is "dumb". The Piksi screams its data over whenever it feels like it. This makes getting any time guarentee about its data incredibly frustrating and unreliabe. When possbile use I2C, or two way master controlled Serial

7.3 Flight Software Layout

Flight Software should be "single threaded". This means that control flow should be a linear decision process, and can be generally characterized by a single state machine. This prevents the headaches of multiple subsystems each owning their own thread and making conflicting decisions.

Flight Software should be centered around a control task. Every given period, a control cycle passes, in which every module of code is executed, and its decisions considersed. Each of these modules are called a control task.

Flight Software should be layered so that control task are bundled, and that decision authority first bubbles up to the highest level control task called the Mission Manager.

The first layer is Monitors. These call driver functions and populate internal memory with the status of sensors and actuator boards. Next are Estimators and Filters. Given previous state information, these perform math computations to gain more information about the spacecaft state over time.

Then are FaultHandlers. These make deducations and conclusions about the spacecraft's health and best course of action given the current and past sensor data.

Then is MissionManager. Using all information available, and the spacecraft's current state, it decides what the next best global policy of the spacecraft should be.

Then are the Subsystem Controllers. Given all data, and the mission state, it decides the best policy for each subsystem. This includes any mathematical calcuations of actuator impulses, torques, etc.

Then are the Commanders. These perform last minute calculations of all the specific settings for a subsystem. This would include pin numbers, addresses to load into registers, thresholds to apply.

Then lastly are the Actuator Control Tasks, which take all those settings and actually dumpt them into subsystems through driver calls.

7.4 Flight Software Implementation

Flight Software should abstract itself into all the layers outlined above so that simulation of its performance can essentially be done by chopping off the Monitors and Actuators, and everything else can be fully verified assuming that the Monitors and Actuators are sufficiently mocked.

As a further upgrade, if these ControlTasks could be simulatenously integrated within PSim, then GNC testing and development need not be translated if PSim and FSW used the same control task architecture.

To minimize the manual linking pain that currently exists within FSW, where programmers have to find field names and match then manually and create 6 different pointers across unit tests and FSW, and auto coder should be used to prevent headaches and prevent errors

For god sake. PLEASE PLEASE PLEASE. The telemetry system of the spacecraft should be completely orthogonal to each ControlTask. Currently internal statefields cannot be looked and or mutated in PTest because they do not have a serializer for the telemtry system. This has caused significant pain.

The tight knit integration between serializers and statefields has made their seperation near impossible, and is incredibly difficult to free. Instead, serailiers should have been specificed in some orthogonal lookup table between statefields and serializers. This way, if a certain test suite does not involve actual flight telemetry, its work is not impeded.

7.5 Flight Software Testing

Testing infrastructure should invest in the capability to "walk" the spacecraft's statemachine all the way to intended testing locations. While this is tough to support, it is aboslutely critical to ensure that the testing conditions that are assumed are actually reachable by the Flight Software binary.