RocketDriver Software Design Document

Spring 2024

Authors

Analicia Sosa

Patrick Clarke

Bonnie White

Reviewers

Astrid Oppen

1.0 Overview

RocketDriver is the software system that controls the physical components of the Theseus rocket and connects those components to the graphical user interface. The primary goal of this software is to ensure safety during testing and launch by collecting temperature and pressure data for operators to monitor, carefully managing actuator and igniter timing, and ensuring the system behaves only in expected ways.

2.0 Design Goals

- Safe system. This software needs to be designed to pass all safety critical software tests.
- Ease of reading and writing. This software needs to be written and structured such that programmers with a level of **XXX** can understand and make changes to the code.
- Fast communication. This software needs to be designed such that its CAN system is able to keep the number of dropped CAN frames low. The functions that write to and read from the CAN bus must also be able to keep up so that the buffers (hardware and software level) do not get full. Keeping the baud rate as high as possible is also desirable.
- Anything I'm missing?

3.0 Milestones

January 20 - 26 — Unit tests planned, skeleton or pseudo code written
January 26 - 29 — Formal schedule for each system
January 30 - February 29 — Weekly code reviews to discuss progress and roadblocks
February 29 — Fully prepared for static fire
March 15 — each burp test planned

4.0 Technical Architecture

The overall plan for developing RocketDriver is split into three components: CAN, peripherals, and the event handler. We shall discuss each one individually here.

4.1 CAN

The CAN bus (Controller Area Network) is how the rocket stand and the Pi Box communicate with one another. Any commands given to the rocket by the user at the gui must go through the CAN bus. The only way that a user can know for sure which state the rocket is in is by the stand sending a CAN message to the Pi Box. Data from sensors, valves, and igniters all use a CAN frame to be understood by the stand and the Pi Box.

4.2 Peripherals

The peripherals of the system are the igniters, the valves, and the sensors. Each of these are physical components. The igniter's job is to start the rocket's engine and to launch the parachute during descent. The valves control the flow of fluid (namely fuel) and the pressure in the rocket. The sensors are responsible for monitoring the physical conditions of the rocket like the temperature and pressure in the fuel tanks.

Because each of these objects exist, and because multiple of each are connected to the rocket, it makes the most sense for this portion of RocketDriver to be built as object-oriented code. Each of the three types of peripherals, (igniter, valve, and sensor) will have their own class. Sensors are differentiated into multiple different types, but we need each to interact with the surrounding system in the same way, so we will build a base sensor class and each type of sensor will inherit from this base class. This will make it easy to add or remove sensor types in the future as well as to modify the numbers of each sensor on the rocket, since they can all be treated as sensor objects.

Each of these objects will be included in a rocket object in whatever quantity is necessary. This rocket object will contain a missionState variable chosen from an enumeration of all possible rocket states. The missionState variable will be responsible for determining whether a given command is valid. The rocket object will also contain a set of timers used to track gui updates and the launch countdown. Creating the rocket class will allow easier modifications to the code in the future and make it easier to understand the code structure.

UML Diagram included in appendix.

4.3 Event Handler

The event handler portion of the code is responsible for ensuring that commands and updates are executed in a timely fashion. The main code operations will be structured to use interrupts to operate. The main types of events the code needs to respond to are as follows: timer interrupt occurs, command recieved, error changing peripheral state, loss of comms.

When a timer interrupt occurs, we want the system to respond by either collecting and sending sensor data or changing the state of peripherals. Sensor data will be sent via routine updates to the gui over the CAN system. The other use for timer interrupts is to precisely time the launch countdown. Imprecision in the launch timing can result in a drastic failure.

The main responses to a command from the gui are: begin timer countdown, change state of peripherals, update configuration files. Commands given by the operators will be used primarily to modify the state of the rocket and move closer to launch. It will also be used to update the configuration of the rocket manually without having to make changes to the code directly. When a command is received, the system will lookup the the requested operation and query the rocket object to determine the validity of the operation. An invalid operation will be sent back to the gui as an error. A valid operation will be executed.

When errors occur in the code, the system needs to respond in a safety-appropriate way and send a message to the gui via the CAN system what has occurred.

When a loss of communication occurs, the system needs to begin a stopwatch timer to track how long the system is without communication and halt any ongoing operations to prevent unexpected behavior.

Interrupts will be split into three categories of priority: safety critical updates, operator commands, background functions. They will be executed in that order of priority. There will exist in code a list of defined command functions that can be easily modified and removed.

5.0 Testing Plan

As with the architecture plan, the discussion shall be broken down by part below.

5.1 CAN

Conducting an initial Benchmark test to determine the existing performance level of the CAN bus should be a priority. The FlexCAN.h file contains tools that can be used for testing. There are tools for determining how full the ring buffer gets (for both receiving and transmitting CAN frames). Ensuring that this stays low will be a priority throughout testing. There are also tools for determining how full the “mailbox” buffers are at a hardware level. Keeping this low should also be a priority (if the buffer gets full at a hardware level the CAN messages are simply dropped/lost). These tools can be used in conjunction to determine how quickly the CAN software is running, and which parts need to be improved. Testing against the original Benchmark in the lab to ensure that improvements are being made.

5.2 Peripherals

Testing of this code will require being in the lab with the ALARA. It needs to deal only with reading from and writing to the various peripherals to the system. The end result of testing in this system needs to perform the necessary data collection and changes to the peripherals and set up sample rocket states and data for the event handler to perform additional tests on. Sensor data from calibrated sensors should be within 95% accuracy of measurements collected manually.

5.3 Event Handler

Since timing is very important in this portion of the system, we will use an oscilloscope to measure timing accuracy and the amount of time taken to handle each interrupt. Testing for this will be impossible to do virtually. It will require use of the ALARA to perform. This portion of testing will likely take more time because of the physical hardware involved in testing it.

It also deals with validating the operations of the system. This portion can be performed virtually, simulating various valid and invalid commands and state movements. We must determine safety-critical operations in the system and test the system to ensure it

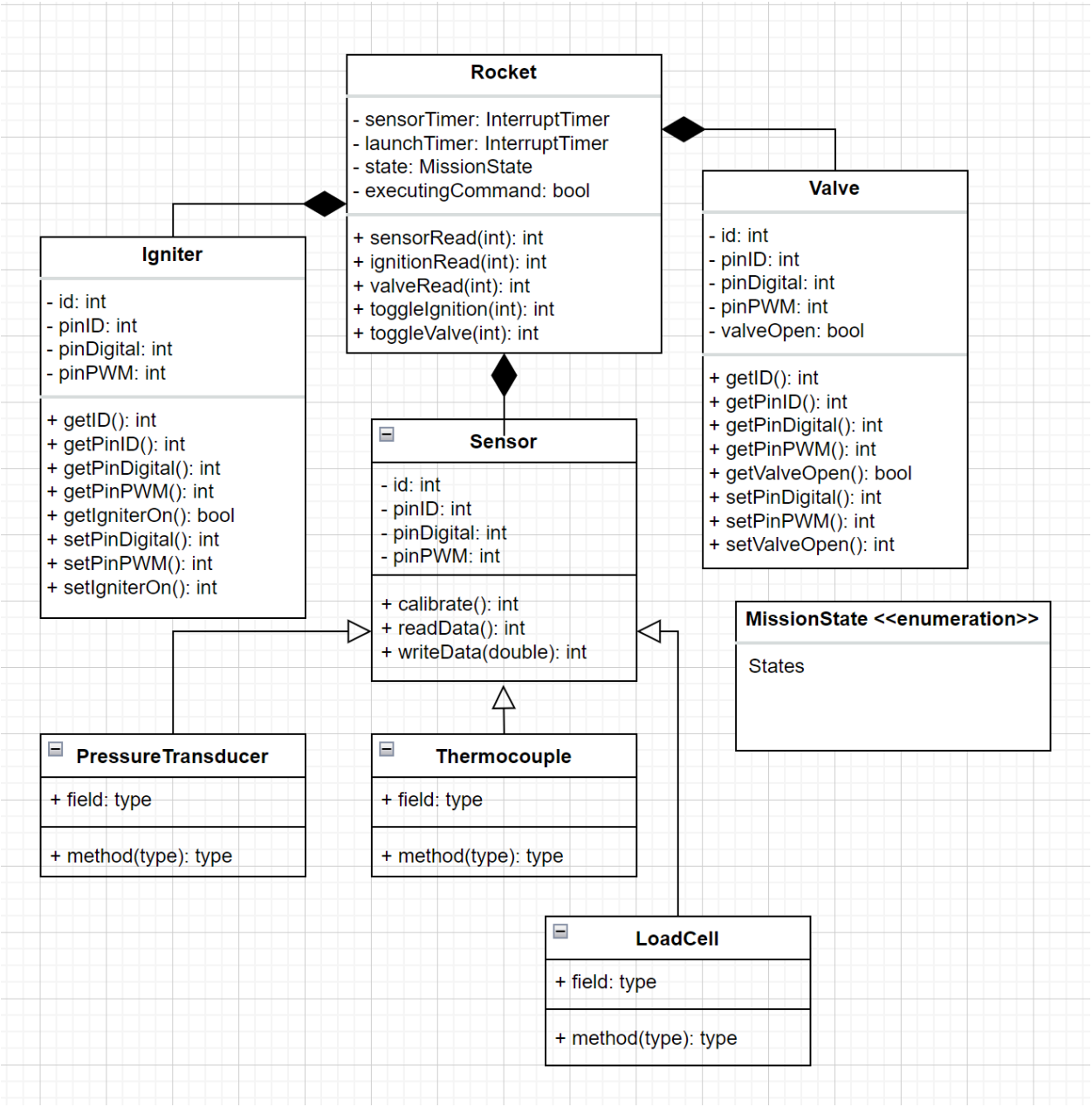
only allows them to be performed under expected conditions. These can be written quickly to test a wide variety of set-ups. All of these tests must pass for the system to be determined successful.

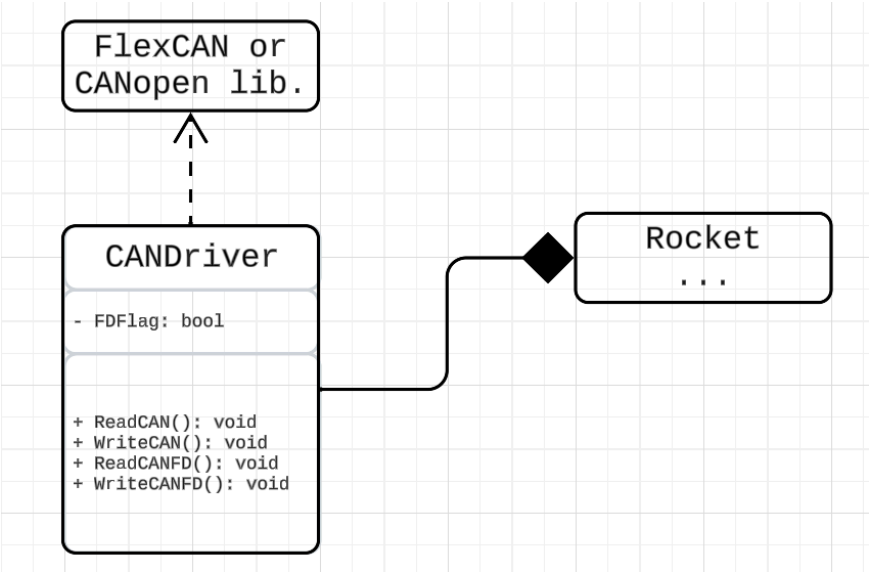
6.0 Deliverables

- Fully functional code
- UML architecture diagram
- UML sequence diagram
- Non-time sensitive items
    - Useful ReadMe for documentation
    - Additional Documentation links / files

Appendix

UML Diagram of peripherals architecture


Rocket
- sensorTimer: InterruptTimer
- launchTimer: InterruptTimer
- state: MissionState
- executingCommand: bool
+ sensorRead(int): int
+ ignitionRead(int): int
+ valveRead(int): int
+ toggleIgnition(int): int
+ toggleValve(int): int
Valve
- id: int
- pinID: int
- pinDigital: int
- pinPWM: int
- valveOpen: bool
+ getID(): int
+ getPinID(): int
+ getPinDigital(): int
+ getPinPWM(): int
+ getValveOpen(): bool
+ setPinDigital(): int
+ setPinPWM(): int
+ setValveOpen(): int
Igniter
- id: int
- pinID: int
- pinDigital: int
- pinPWM: int
+ getID(): int
+ getPinID(): int
+ getPinDigital(): int
+ getPinPWM(): int
+ getIgniterOn(): bool
+ setPinDigital(): int
+ setPinPWM(): int
+ setIgniterOn(): int
Sensor
- id: int
- pinID: int
- pinDigital: int
- pinPWM: int
+ calibrate(): int
+ readData(): int
+ writeData(double): int
MissionState <<enumeration>>
States
PressureTransducer
+ field: type
+ method(type): type
Thermocouple
+ field: type
+ method(type): type
LoadCell
+ field: type
+ method(type): type


FlexCAN or
CANopen lib.
CANDriver
- FDFlag: bool
+ ReadCAN(): void
+ WriteCAN(): void
+ ReadCANFD(): void
+ WriteCANFD(): void
Rocket
...