Hyperloop Manchester Specifications, Architecture, System Design and Testing Procedures

Harry O'Brien, Lead Developer & Head of Electronics

May 24, 2020

Contents

1 Introduction

This document outlines the structure for Hyperloop Manchester's digital components through increasing levels of detail starting with a list of specifications, to listing, all required classes, methods, functions, variables and the details of how sub-modules may interface between each other.

If this document needs to be modified, please do so through a merge request on gitlab.

2 Specifications

The following specifications are the most basic requirements of what the Hyperloop pod must do. Here we set out everything that relates to the software and electronics.

2.1 Sensing

1. The system must be able to know its location (relative to the length of the Hyperloop track) with a tolerance of less than 0.2m.
2. The system must be able to know its distance from the rails with a tolerance of less than 1mm.
3. The system must be able to validate that a sensor is alive and performing as intended.
4. The system must be able to show the properties and status from any selected sensor.
5. The system must be able to show, in real time, readings from any selected sensor.
6. The System must be able to detect and try to correct any erroneous sensor readings.

2.2 Dynamics

1. The system must be able to dynamically brake depending on its environment and current location in order to maintain control.
2. The system must accelerate depending on its environment and current location so that the highest possible speed is achieved.
3. The system must be able to perform an emergency stop when requested.
4. The system must be able to reach and maintain a selected speed quickly and reliably.
5.

2.3 Communication

1. The system must be able to perform wireless communication with an external device.

2. The system must have some form of remotely-connected GUI that allows users to modify internal system settings.

2.4 Navigation

1. The pod must be able to travel to a given location on the Hyperloop track unsupervised and arrive safely at its destination.c+

2.5 Logging

1. The system must log all key events such that simulated replay is possible.
2. Log when and why a process exits.
3. Log when and how long it took for a given process to start up.
4. Log notable events during the operation of the software.
5. Regularly log sensor data at a rate of 10 Hz.

2.6 Remote Access

1. The system must be able to receive new firmware and software over the air.
2. The system must be able to read and write settings to a permanent location so that any changes can be recovered on system restart.
3. The system must maintain a constant state between all GUIs that are connected simultaneously
4. The system must continue stop if a user unexpectedly disconnects

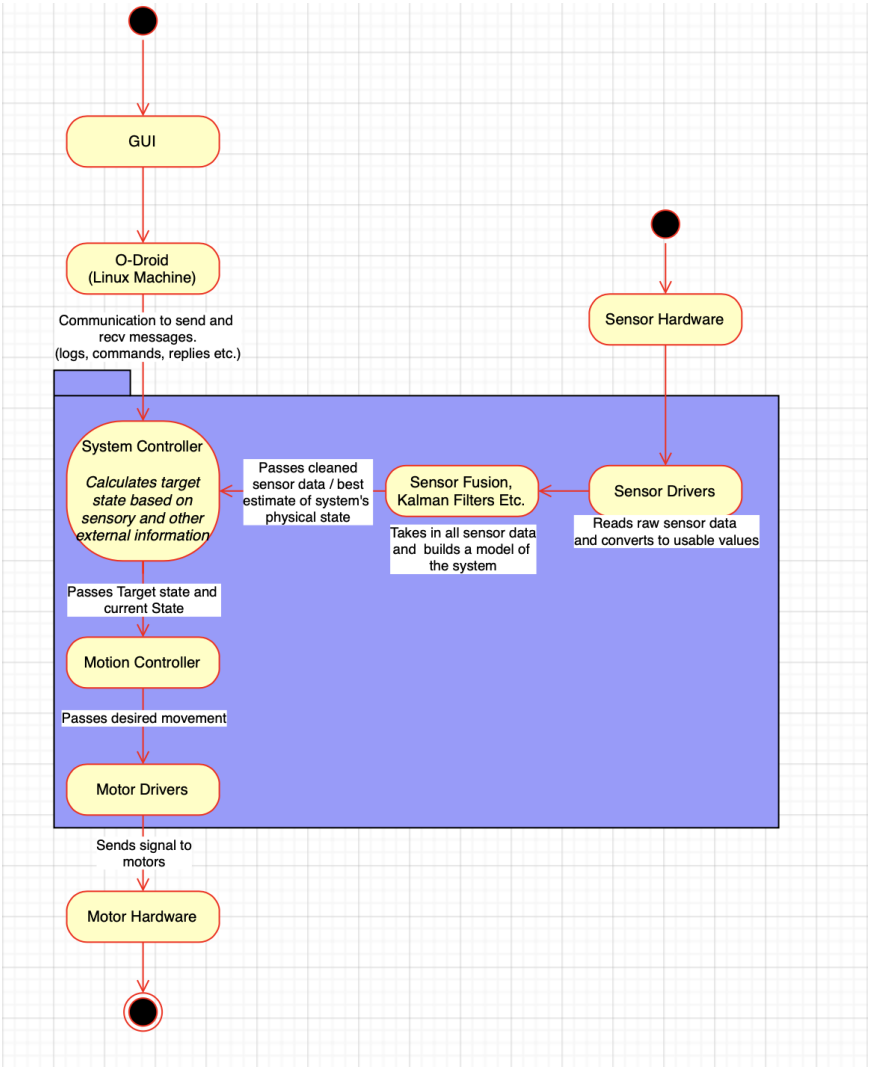
2.7 System Reliability

1. All services must be constantly up - any abnormal exits from a program must be logged and restarted.
2. The system must recover from a misbehaviour in the software/hardware.

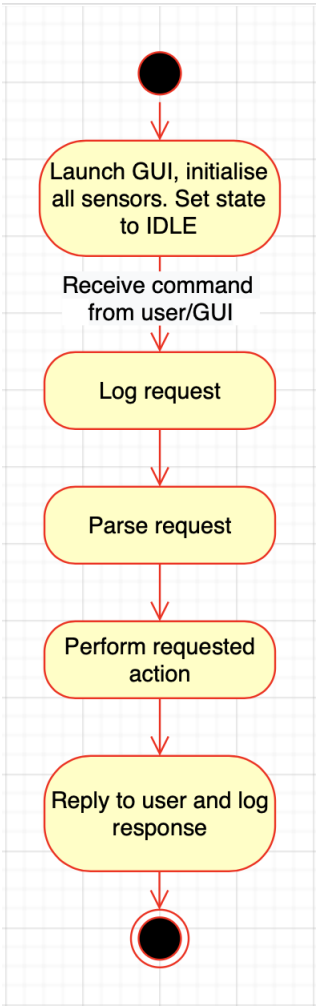
3 System Architecture

Here, we are documenting the system architecture via a collection of block diagrams, with directional arrows connecting subsystems. We identify how data flows and show partitioning at the hardware vs. firmware level. All entities inside of the blue box is software contained on the 'Teensy' Micro Controller.

3.1 System Flow Diagram



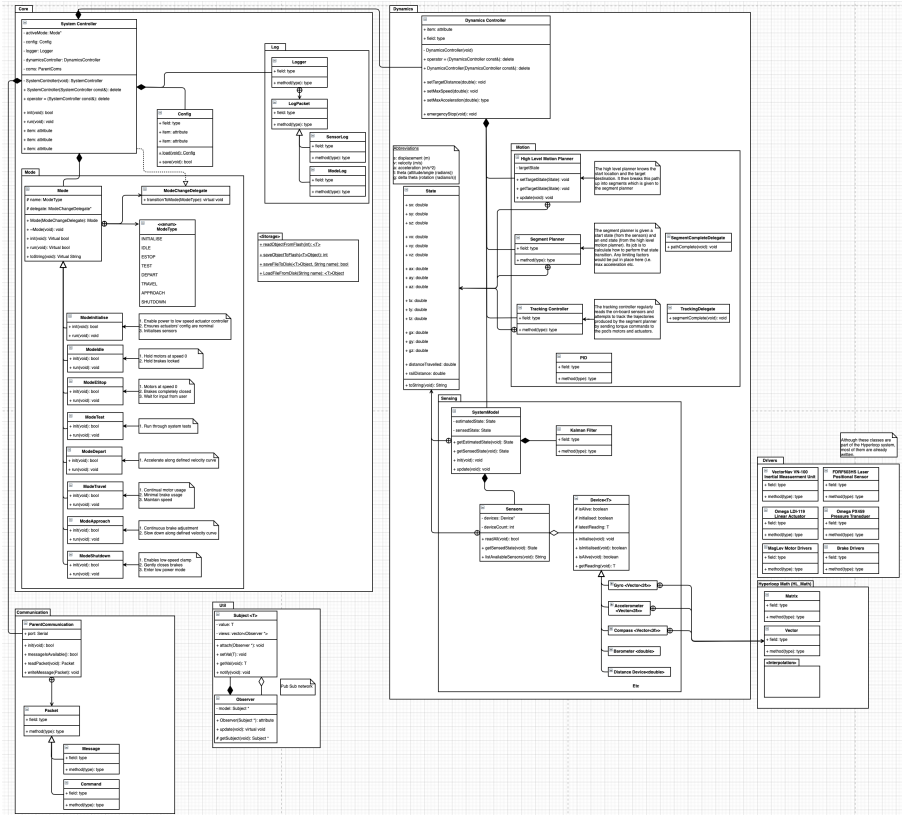
3.2 External Message Flow Diagram



4 System Design

This section describes the firmware design. This includes names for all required methods, functions and variables. In addition to naming, this section contains details of interfaces between subsystems.

4.1 Class Diagram



5 Testing

Every embedded system needs to be tested. Generally, it is also valuable or mandatory that testing be performed at several levels. The most common levels of testing are:

- System tests verify that the product as a whole meets or exceeds the stated requirements. System tests are generally best developed outside of the engineering department, though they may fit into a test harness developed by engineers. **The system tests should treat the firmware as a black box.**
- Integration tests verify that a subset of the subsystems identified in the architecture diagrams interact as expected and produce reasonable outcomes. Integration tests are generally best developed by a testing group or person within software engineering.
- Unit tests verify that individual software components identified at the intermediate design level perform as their implementers expect. That is, they test at the level of the public API the component presents to other components. Unit tests are generally best developed by the same people that write the code under test. **Proponents of test-driven development advocate that tests at this level be written in advance of the functions or classes that they are intended to verify.**

Of the three, system tests are most easily developed, as those test the product at its exposed hardware interfaces to the world (e.g., does the dialysis machine perform as required). Of course, a test harness may need to be developed for engineering and/or factory acceptance tests. But this is generally still easier than integration and unit tests, which demand additional visibility inside the device as it operates.

To make the development, use, and maintenance of integration and unit tests easy, it is valuable to architect the firmware in a manner compatible with a software test framework. The single best way to do this is to architect the interactions between all software components at the levels you want to test so they are based on publish-subscribe event passing (a.k.a., message passing).

Interaction based on a publish-subscribe model allows a lightweight test framework like the one shown in Figure 2 to be inserted alongside the software component(s) under test. Any interface beween the test framework and the outside world, such as a serial port, provides an easy way to inject or log events. A test engine on the other side of that communications interface can then be designed to accept test "scripts" as input, log subscribed event occurrences, and off-line check logged events against valid result sequences. Adding timestamps to the

event logger and scripting language features like delay(time) and waitfor(event) significantly increases testing capability.

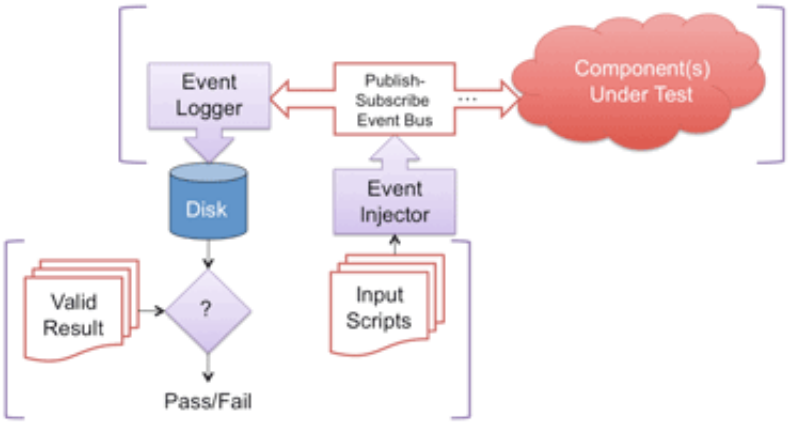


Figure 1. A Test Framework Based on a Publish-Subscribe Event Bus

It is unfortunate that the publish-subscribe component interaction model is at odds with proven methods of analyzing software schedulability (e.g., RMA). The sheer number of possible message arrival orders, queue depths, and other details make the analysis portion of guaranteeing timeliness difficult and fragile against minor implementation changes. This is, in fact, why it is important to separate the code that must meet deadlines from the rest of the software. In this architecture, though, the real-time functionality remains difficult to test other than at the system level.

6 To Do

To-do list for this document:

1. Convert flow diagrams from images into compiled Latex
2. Build out class diagrams
3. Create architecture for O-Droid device
4. Create class diagrams for O-Droid
5. rewrite testing to make more sense in this specific application