Aero FC System Release Version 1.0

System Architecture Design Document

Brandon Braun, May 2019

Table of Contents

[Broad System Design Goals 4](#_Toc7638109)

[Hardware Architecture 4](#_Toc7638110)

[Processor 4](#_Toc7638111)

[Sensors 4](#_Toc7638112)

[External Memory 4](#_Toc7638113)

[System IO 4](#_Toc7638114)

[Software Architecture 4](#_Toc7638115)

[Language Selection 4](#_Toc7638116)

[Design Paradigm 4](#_Toc7638117)

[Criticality 4](#_Toc7638118)

[Abstraction 4](#_Toc7638119)

[Isolation 4](#_Toc7638120)

[Core System Functionality 4](#_Toc7638121)

[State Predictor 4](#_Toc7638122)

[Control Law 4](#_Toc7638123)

[Data Logging 4](#_Toc7638124)

[Asserts 5](#_Toc7638125)

[User Input 5](#_Toc7638126)

[Resource Protection/Isolation 5](#_Toc7638127)

[Hardware 5](#_Toc7638128)

[Software 5](#_Toc7638129)

[System Dependencies 5](#_Toc7638130)

[Testing 5](#_Toc7638131)

[Criticality Level 5](#_Toc7638132)

[Low Level Hardware Drivers 5](#_Toc7638133)

[Component Modules 5](#_Toc7638134)

[Integration Tests 5](#_Toc7638135)

[Continuous Integration 5](#_Toc7638136)

[Compile Jobs 5](#_Toc7638137)

[Test Jobs 5](#_Toc7638138)

[Alerts 5](#_Toc7638139)

# Broad System Design Goals

System Release Version 1.0 is the first attempt at creating an infrastructure robust and capable enough to support a highly feature extensible drone flight controller system. For the current design iteration, this will look like a no-frills design that is capable of controlling a drone in nearly level flight conditions, with minor perturbations from the user controller. While this is technically a primarily software oriented design project, there are some other considerations that should be met as well, mostly in regards to testing, building, and quality assurance.

# Hardware Architecture

Seeing as this system will be initially created on the development board I designed in graduate school, some of the development hardware specifications have already been decided for me. Several features though are going to need some hardware changes, so a new board will be in the works while writing the software.

## Processor

The current processor will be the STM32F446RE, which is an ARM Cortex M4 device with 128k of SRAM and 512k of Flash. Some useful aspects of this device is a multi-channel DMA controller and a hardware floating point unit that will speed up some of the mathematical calculations.

## Coprocessor

There is no coprocessor support planned for this release, but there are plans in the next iteration to add a small FPGA that will (hopefully) speed up some of the mathematics used in state estimation, filtering, and system control laws.

## Sensors

For this release I will be working with the bare minimum for flight, a single 9-DOF accelerometer, gyroscope, and magnetometer (LSM9DS0). This provides enough information for an adequate system state to be generated via the state estimator.

## External Memory

There will be three sets of external memory: uSD, NOR, and FRAM. The uSD device will be used for dumping large amounts of information pertaining to non-critical systems and will not be considered power failure safe. The NOR flash will be used to log power safe critical and non-critical data that does not frequently change. The FRAM device will be used for information that will change frequently, whether that is at runtime, between power cycles, or simply needs to be updated very quickly.

## System IO

At the moment, because the system complexity is still quite low, the number of system input channels is sparse. At the time of writing, this only consists of a USB-to-Serial adaptor for connecting with an external PC and a UART channel for the Radio interface.

## System Programming

For the first iteration of this software, all the programming will be done via a JTAG interface. In future variants, a bootloader will be created so that the software can be updated over a serial port or loaded from the uSD card.

# Software Architecture

## Language Selection

The language of choice for this system will be C++17 as it provides a good mix of modern development tools and features while still allowing code to run at a fairly low level for access to things normally encountered in embedded systems. All new code will be generated using this standard and interface with various C libraries appropriately as needed.

## Compilers

The compiler of choice for the embedded system is the GCC [arm-none-eabi](https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads) compiler, whatever is the latest release at the time. All of these will be supported by the Boost-Build system natively.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Embedded System** | **Windows** | **Linux** |
| **Compiler Of Choice** | GCC arm-none-eabi | MSVC | GCC |
| **Version** | 6 or greater | 1910 or greater | 7.4 or greater |
| **Notes** | [Download](https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads) |  |  |

## Design Paradigm

### Criticality

The software shall be layered into design criticality levels, each indicating the severity of an issue if something in that software level should fail. These are taken directly from DO-178B and are restated as follows:

**Level A** - Catastrophic: Prevent continued safe flight or landing  
**Level B**- Hazardous/Severe-Major: Potential fatal injuries to a small number of occupants  
**Level C** - Major: Impairs crew efficiency, discomfort or possible injuries to occupants   
**Level D** - Minor: Reduced aircraft safety margins, but well within crew capabilities   
**Level E** - No Effect: Does not affect the safety of the aircraft at all

Most of the core functionality that will be designed will lie in levels A and B for this release cycle.

### Abstraction

A key design component that I’ve found in several of my other projects in the past is designing parts of the system to operate to an Interface of some kind or another. This allows complex components to be abstracted out of the way and present a consistent interface to other parts of the system. In addition, should a component need to be updated in the future, it can be done so with relative ease assuming it does not change the behavioral interface. Wherever possible, system functionality will be abstracted into some kind of interface, whether that is a C-style module or a modeled inside of a class.

### Isolation

Wherever possible, each subsystem or module shall be created with the intent that it implicitly knows nothing about the other code executing on the flight controller. Should it need to utilize information from another system or perhaps inform a different thread of an event, it will need to go through the proper channels in the flight controller kernel.

## System Kernel

### Parameter Manager

The parameter manager is responsible for managing access to pieces of information that the system as a whole will have access to. This is usually POD types or some kind of custom data structure. Threads can request/update the data on a registered parameter and have it returned to them without any worry about race conditions. Each parameter will have various properties and actions associated to it that will alter interaction behavior. For example, a parameter could be read, updated, locked, unlocked, etc. The full implementation of this will be decided later in the module’s design document.

### Event Manager

The event manager is responsible for propagating system events throughout the code. This shall be accomplished through the use of data producers and data consumers, each registerable with the core kernel module. There can be multiple producers for a particular kind of event as well as multiple consumers, effecting making this a Multiple Input Multiple Output (MIMO) system. Another protocol very similar to this is the MQTT pub-sub model.

### Log Manager

The log manager is responsible for routing logging information of all kinds into a data sink. This provides an outlet for information that does not need to be passed around the system but still needs to be recorded somewhere, typically for a human to look at later. The sinks can be registered into the logging framework and its behavior highly configurable. A potential candidate for this system is [spdlog](https://github.com/gabime/spdlog), which claims to be a lightweight logging framework. The particular sinks that may be useful for this release are a UART sink for printing information to a terminal and a Non-Volatile Memory sink for long term storage of something like data logging or system asserts.

## Core System Functionality

### State Predictor

This module is responsible for talking to the IMU device to pull accelerometer, gyroscope, and magnetometer readings and transform them into an estimate of the drone’s current orientation. This data will then be pushed through the parameter manager for other systems to utilize.

### Control Law

Responsible for generating ESC control/error signals that correct the system’s orientation to match a given reference value. All input and output data shall pass through the parameter manager.

### ESC Controller

Responsible for translating error signals from the Control Law system into something that the ESCs can understand.

### Data Logging

The user will likely want to record flight data, so a system shall be put in place that reads the parameter manager at a specific interval and log key pieces of information to a non-volatile sink registered with the Log Manager.

### Asserts

Asserts are a fantastic way to track down asynchronous events in code that are difficult to catch inside a debugger. As such, there shall be another system that registers a sink with the Log Manager to dump asserts into non-volatile memory.

### User Input

#### Stick Inputs

The system shall be capable of taking input from an external radio receiver so that the user can send inputs that direct the orientation of the quadrotor.

#### Command Line Inputs

The system shall be capable of parsing inputs from an external device over a UART/USART channel to respond to various command line style inputs. This is intended to allow a user to request information from the system or update available configurable parameters.

#### Button Inputs

Optionally, the system could have inputs from one or more on board buttons. The system shall be capable of attaching functional behavior to a press event. It is anticipated that these buttons will be used to transition through operational modes or allow manual adjustment of system parameters.

# Testing

## Low Level Hardware Drivers

Hardware drivers are a tricky beast to test. Due to the anticipated large time sink associated with this (writing tools and developing hardware), I believe I will be leaning more towards debugging the hardware as the problems arise. Currently I’ve got a pretty decent set of core code for my most used peripherals and I feel confident enough in them that I can build on top of it.

## Component Modules

These are tests whose scope is intended to be limited to one compilation unit at a time. The primary goals here are to achieve coverage for the module under test as well as verify functional behavior according to the design goals.

## Integration Tests

Higher level integration tests will be performed to ensure that different systems are able to communicate effectively and operate as intended.

# Continuous Integration

Continuous integration with Jenkins shall be utilized to ensure changes to the code base do not cause major issues.

## Compile Jobs

Compile jobs shall only be responsible for compiling system code and running static analysis tools to ensure that no major errors were presented.

## Test Jobs

Test jobs shall be responsible only for executing tests on system code at the module level. Any test failures will mark the entire job as failed.

## Integration Jobs

These jobs will be specially designed to run large scale integration tests between higher level systems, once that becomes feasible.

## Alerts

When a build fails, an email will be sent to let the user know that things went awry.

## Pipeline

In keeping with the modern times, it would be great to have a pipeline setup that will automatically produce release binaries that can be directly loaded on to the system assuming that all tests and other checks pass with flying colors. Ideally this would be a single button press and at the end, a finished binary is archived.

# Software Management

## Branching

## Issues

## Reviews

Design review checklist. Submitted with each ticket???

## Documentation

## Tracking Progress

GitHub