# Design Notes for the E-Vtol Simulation Project

Kodiak North

## 10/09/2024 – Initial list of notes/tasks/questions to guide the design

* First and foremost, I’ll need to configure VS Code to compile and run C++ executables.
* What are some unit tests I can incorporate into the code?
  + The first idea that comes to mind is a test to ensure all 5 vehicle types end up in the random list of 20 every time it is created.
  + I can also design a test to remove battery life from the equation and ensure each vehicle type will fly the correct distance given 1 hour of simulated time.
  + Similarly, I can test that the batteries drain in the right amount of time.
* I should obviously create a parent class to define an e-vtol vehicle with the provided characteristics, and then child classes for each specific vehicle type will handle the intricacies between different velocities, number of passengers, etc.
* How will I handle the possible generation of a fault?
  + A 25% chance of a fault per hour does not necessarily mean that a fault will occur every 4 hours!
* So the simulation does not actually take 3 hours, 1 loop step can be considered 1 second, or even 1 minute of elapsed time.
* How will the equation for energy consumption work?
  + Say cruise speed is 100 miles/hour, battery capacity is 50 kWh, and energy use at cruise is 2 kWh/mile…
  + How long the battery will last flying at cruise speed?
    - 50 kWh / (2 kWh/mile) = 25 miles… => 25 miles / (100 miles/hour) = 0.25 hours… => 0.25 hours \* (60 min/hour) = 15 min.
  + So the equation is:
    - ((Battery capacity [kWh] / energy use at cruise [kWh/mile]) / cruise speed [miles/hour]) \* 60 [min/hour] = duration in minutes
    - Will have to incorporate the loop step into this! Or perhaps this value could be stored for each vehicle type. As the system loops, it increments a counter. Once the counter equals the calculated duration, the battery is dead.
* Design a state machine for each vehicle. Upon initial thoughts, the states could be:
  + Idle, flying, charging, or waiting to charge
* Will need create a queue to hold vehicles that are waiting to charge, and also track chargers that are available or not.
* How will I assign a vehicle to a charger? It can probable be as simple as chargers 1, 2, and 3 are in the queue of available chargers. When a vehicle needs a charger, it is dequeued and its ID populates the ‘current charger’ member variable for that vehicle. Once charging is complete, the charger’s ID is enqueued back into the queue of available chargers.
  + If the queue of available chargers is empty, that means there are no available chargers and a vehicle will have to be placed in line.
* As for recording metrics:
  + If I track the flight time, I think I’ll know each vehicle’s distance travelled based on the cruise speed.
  + I can record vehicle charge time and store it in a member variable.
  + Store number of faults in a member variable.
  + Total passenger miles can likely also be calculated at the end of the simulation based on flight time.

Okay! So this is a lot of information spewed out. It’s time to organize it into a list of priorities, and concretely define the data structures that I’ll use.

## 10/10/2024 – The Beginnings of a Plan

10/11/2024 – updating the plan starting at 5a.

10/12/2024 – brainstorming about random number generation starting in 7a.

1. ~~Install C++ compiler and build a simple hello world application to make sure the toolchain is squared away.~~
2. ~~Define the data structures for the E-Vtol parent class, and one specific vehicle type child class. Let’s just start with the Alpha Company.~~
3. ~~Create one instance of an Alpha aircraft and perform some tests!~~
   1. Will need the main loop to step through time. Can initially loop the corresponding number of times for 1 simulation hour. This will be 60 times if the desired loop step is 1 minute.
   2. Start in the idle state, move to the flying state, fly until the battery is dead, and then return to the idle state.
   3. Record metrics during the flight, and verify that everything looks correct at the end! At this point in time, we’re not worrying about charging or faults. Data such as flight time, distance traveled, and total passenger miles should be easy to test now.
      1. The expected flight time for the Alpha aircraft is 100 minutes using the equation calculated yesterday. If the loop is kept at 1 hour for this first test, the Alpha plane should still be in the flying state at the end of the loop, with 60 minutes of recorded flight time. This will be an excellent test to perform!
      2. If all passes, then loop for 2 hours, and expect the state to be idle at the end with 100 minutes of flight time.
4. ~~Now we have an aircraft that can fly until its battery dies, and then stop flying. It would be a good time to incorporate one charger for our one aircraft to recharge with, and record its total time spent charging~~
   1. With only one charger, we don’t have to worry about creating the queue right away.
   2. When the battery dies, the Alpha plane can be placed directly into the charging state.
   3. It will take 36 minutes to recharge its battery.
   4. For our 1-hour test, there should be no recharging.
   5. With the 2-hour test, the total time spent charging should be 20 minutes, and the final state will be charging at the end of the loop.
   6. It’d be good to now perform a 3-hour test. What will we expect to see then?
      1. 180 min total. 100 spent flying, 36 spent charging, and then 44 spent flying after. So 144 total min flying.
5. ~~With one vehicle flying, recharging, and flying again; we’re at a very good starting point. Now will be a good time to incorporate another vehicle and another charger. How will we have to modify the code to implement this?~~
   1. The chargers in use can actually be simplified to just a number, rather than a queue! Increment the number as chargers are put in use, and decrement as they become free. Very simple.
   2. As for adding a second plane, let’s throw the Bravo company into the mix. We should perform similar tests as described in (3) and (4). Do I have to recalculate all of the expected values for Bravo or is there a simpler or automated way to do it? Once I figure that out, it will be easy to make a small array of the two vehicles and loop through it per each step of the main simulation loop.
   3. 10/14/2024 update – deciding to add Delta company second because it is slightly more complicated with its recharge time
      1. 1 hour test – 60 min of flight, no recharging
      2. 2 hour test – 100 min of flight, 20 min of recharging
      3. 3 hour test – 142.8 min of flight, 37.2 min of recharging
6. ~~Once Bravo company is tested and working well. Add one of each of Charlie, Delta, and Echo.~~
7. ~~Then we can make a bigger array and test multiple of each of the aircraft in the mix.~~ ~~The random generation of some amount of each plane can be done at the end.~~
   1. Initially, I’m thinking about generating a random number between 1 and 5, which will be used as the ID (Alpha, Bravo…), and also select the default parameters (cruise speed, battery capacity…) to supply to a constructor and dynamically create instances of each aircraft type.
      1. If the random number generation is truly random, it’s possible that one aircraft type doesn’t make it into the list of 20. I can fill the first 5 list items with aircraft ID 1-5 to guarantee that all make it in the list, but that will affect the randomness of the simulation… maybe I put them at the end?
   2. As I start to define the e-vtol class, think about what data is common for every object, and what data is unique to each specific object. This may help with saving memory. For instance, every Alpha aircraft has the same cruise speed, battery capacity; but each aircraft will have a different total flight time based on when they run out of battery, how the charging order works out.
      1. With the running flight time per vehicle. If that flight time modulo duration per recharge == 0, it’s time to charge.
8. ~~Similarly, fault generation can be thrown in at the end. I could even start simple and increment a counter somewhat randomly, and then work in the actual percent chance per fault later.~~
   1. I’ll assume that a fault can occur at any time; whether flying, charging, or waiting to charge.
9. Also, remember that you’ll eventually want to write recorded data into a file.

## 10/11/2024 – Calculations of flight times and recharge times per vehicle type

10/12/2024 – put more thought into the loop step now that we must deal with tenths of seconds.

**TODO** forgot about the queue of vehicles in line to charge. Let’s think of that tomorrow, and possibly how to simplify it!

Finish the table tomorrow, and then start to come up with a concrete definition of the aircraft class.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| Flight time [min] | 100 | 40 | 37.5 | 100 | 51.72 |
| Recharge time [min] | 36 | 12 | 48 | 37.2 | 18 |

**OOF** the 37.2 minutes will make it tough to have a loop step of 1 min without using floating point values… perhaps step in seconds. Or I could do something for Delta where every 5 loop steps, 1 minute is added to the recharge time, or however that needs to work. I could also treat a loop step as 0.2s. As of now, that appears to be the finest resolution I’ll need. Let’s wait on populating the rest of this table in case anything else changes. In the table, calculations of time in minutes will be have to be multiplied by 5 if the loop step is 0.2s.

**10/14/2024 update –** I jumped the gun. Should have calculated the whole table before choosing a loop step. Echo’s flight time is what should be driving the loop step decision. Luckily the code has been nicely designed to handle a change in loop step! I’d like to try to be accurate to the 100th’s of a second. This will affect performance because we’re looping so much more. As long as it’s not horribly slow with all 20 aircraft in the mix, let’s stick with it. Otherwise, we can always change the loop step to 10th’s of a second.

Let’s round off Echo’s flight time to 51.72 min, and choose a loop step of 0.02 seconds. **10/16/2024 update** – the 0.02 second loop step does not bog down the program. Let’s stick with it.

## 10/12/2024 – Queue of planes in line to charge

So, we’ll eventually have an array of 20 aircraft. When an aircraft becomes ready to charge, we can simply enqueue its index. This will maintain a lightweight queue of uint8’s.

## 10/12/2024 – Aircraft class

Common data to a particular vehicle type:

* Cruise speed, battery capacity, time to charge, energy use, passenger count, fault probability
  + These can be condensed. Many will be used to calculate what’s really important…
    - Flight time until battery dies (loop step ticks)
    - Time to charge (loop step ticks)
    - Passenger count
    - Fault probability

The data listed above does not have to be a member variable for every single object, because, for example, every single Alpha aircraft will have the same values. I could store this data outside of the class and access it using the aircraft ID, but this would be slightly messier at the expense of saving some bytes. It might be nicer to store it in the class’s member variables. This will end up being a game time decision.

Specific data to each vehicle:

* Aircraft ID
* Current state
* Time spent in the air
* Time spent charging
* Number of faults

I think I’m finally ready to start writing code now. Woo!

## 10/14/2024 – More detailed plan for greater than 3 planes in the mix

The plan so far has worked out well. All 5 plane companies have been tested, BUT only 3 at a time so there was always a charger available. Now it’s time for things to get more complicated.

With a fourth plane added to the simulation, it’s possible that a plane will have to wait in line for a charger. Creating the queue to place a plane in line will not be difficult, and the SW implementation of dequeuing a plane in line once a charger frees up will also not be difficult.

The important question here is what can we do to test and verify that things are working correctly once a fourth plane is added?

With the 1, 2, and 3 hour tests passing for all aircraft, we have verified that the accumulation of flight time, and charge time are working correctly when a plane is either in the flying state or charging state. We’ve also verified that the transition from flying state to charging state, and back is working correctly.

So, we really only have to test the queue, and enqueuing/dequeuing planes at the right time.

Oh, C++ comes with a built-in queue library. That’s very nice!

## 10/16/2024 – List of assumptions made

* Initialize all aircraft into the flying state at t = 0. Simulate from t = 1 min to t = 180 min to run for a full 3 hours worth of simulation time.
* A fault can be generated regardless of what state the aircraft is in.
* For recording data, the “average flight time per flight”… “per vehicle type” is confusing the brain. If the simulation is run for long enough, the flight time per flight would be the time it takes the battery to drain. But since the simulation is relatively short, many flights will get cut short. This must be the purpose of the exercise.
  + For each vehicle, I’ll record total flight time, and total number of flights.
  + Then for each vehicle type in the simulation, I’ll add up the total flight times and total number of flights to compute the average for that vehicle type.
    - Average distance traveled per flight is the average flight time per flight multiplied by the vehicle type’s cruise speed.
  + I’ll do the same thing for average time charging per charge session per vehicle type.