Analysis of Solution

My solution has five user defined tasks: Physics Task, Throttle, Angle Setpoint, LCD Display, and an Idle Task. These tasks and their relative priorities can be seen in the screenshots "segger_tasks0.png" and "segger_tasks1.png" these also show the kernel defined tasks that were relied on such as the Kernel's Timer Task. The highest priority task is the Physics Task, it is the bulk of the computation required for gameplay, while the lowest priority task (other than Idle) is the LCD Display because it was decided that a little lag in display would be better than displaying an incorrect output. The three tasks that are triggered periodically (physics, throttle, and LCD) all have the same period because it is the smallest period the OS software timer allows. Therefore, each time these tasks run they are all scheduled at nearly the same time creating a situation just shy of a critical moment. However, as seen in "segger_tasks1.png" they consistently meet their deadlines and allow for the Idle task to run often.

Memory usage was as follows:
Flash used: 83728 / 1024000 (8%)
RAM used: 35396 / 56000 (63%)

This project was not the lengthiest code I have written, it fit in a moderate number of files with reasonable line counts so it makes sense that not much flash was used. Still, it uses a lot of micrium OS functions and variables and with up to six software timers running at one time it makes sense that the RAM usage was much higher.

The physics engine was able to run at the lowest period I could set with the OS software timer and it still finished early enough to allow time for the Idle task. Creating the kinematic equations in code required thought, the force and acceleration are memoryless. They depend only on the thrust right at that specific moment. The velocity and position are not memoryless however, and depend on the previous values times a time delta. The time delta was set to the value of the tasks period and on each update velocity was calculated as the previous velocity plus the new acceleration times the time delta. Then we take the old position plus the new velocity times the time delta to get the new position. I am also proud of how the blackout duration was handled: if blackout occurs a flag is set saying we are in blackout mode and the user cannot thrust anymore, then we kick off a software timer for the blackout duration and that timer will turn off the flag when it completes.

The config variables were set to be playable while mimicking reality, for example gravity is at 10m/s^2 (nearly earth's 9.8m/s^2) and the rocket has a mass of 6000kg. The distance unit I worked with was meters, so the lcd was 128m wide. The playability depends on the mass to gravity relationship as well as the conversion efficiency. The max fuel drain is 40 kg of fuel so

40 times the conversion efficiency divided by mass should be greater than or equal to the acceleration due to gravity so the ship can counteract gravity at full throttle.

40*conv_eff / mass >= gravity

Outside of this bound the ship may fall too fast to be able to play it or it may not fall fast enough to make it entertaining.

Given more time I would build on the configurability of the project. At one time I had hoped to read in the configuration parameters from an external file or make a welcome screen where a player could edit certain parameters using the buttons. I started the development to take in a file of config information but parsing the text became too big of a task and I decided to cut it--strings in C can quickly become a hassle. With more time I would have continued on that path.