**Additional Software Design Commentary**

Rust has native support for async/await a la JavaScript Futures, but relies on executors design for specific targets. Our codebase relies on Embassy, which targets embedded platforms and has first-class support for our STM32 development board. Rust’s async functionality works like this:

1. Define a function using the async keyword. This function can take any arguments and can either be a normal function that runs top to bottom, never returns, or returns a future.
2. Perform any setup. For us, this is initializing peripherals to required states, configuring clocks, and generating peripheral handles.
3. Use the executor to spawn new tasks, passing our async functions to the executor. The executor will immediately perform these tasks at the specified priority until they yield execution through calling await on a future.

After this, the executor essentially swaps between the main event loop and whichever async functions interrupt execution. This model fits the embedded paradigm well, forgoing hardware interrupts for built-in language features. This made our code much easier to develop and debug after overcoming initial hurdles of understanding an unfamiliar execution model

**Design Process**

The past few years of software development on our team has been scattered and stagnant. At the beginning of this season, we had a four-year-old codebase written by a single person with no documentation and no transfer of information from the original author. Subteam membership hadn’t been consistent enough to have a direct succession of programmers from the authors of the previous iteration. With this in light, combined with difficulties in building the old codebase and satisfying dependencies, we ultimately made the decision to attempt to rebuild the MP6’s software requirements from scratch.

We identified the following goals and constraints for our ideal software stack:

1. Asynchronous execution model: The control loop in the vehicle is event driven, not synchronous. The runtime is going to be sitting and waiting for events to process, i.e. throttle input, AMS monitoring, and CAN communication to the electric motor controller.
2. Ergonomic API design: abstracting away component interaction into API calls, so that future developers could refactor code more easily for potentially different hardware, as well as allowing for less experienced developers to use device drivers without being bogged down by implementation details.
3. Descriptive and complete documentation: the MP5 codebase was essentially impossible to parse and understand. Effective and complete documentation makes the codebase have a higher longevity.

With these in mind, we initially chose to design the new application stack using ST Microelectronics’ vendor-provided hardware abstraction layer, combined with Microsoft Azure RTOS to satisfy our requirement for multiprocessing.

As the season progressed, however, the number of design hours available to the subteam decreased significantly as the academic year progressed. Particularly, issues with the vendor provided toolchain and general inexperience with multiprocessing and Azure RTOS slowed development significantly. Oftentimes, programming felt more like fighting the compiler and toolchain rather than figuring out the implementation. Eventually, due to the combination of inexperience, lack of time/manpower, and hardships with the toolchain, development essentially halted.

We decided halfway through the season to officially pause development on the codebase and try to build out a minimum viable product using Rust and Embassy. After overcoming the initial difficulty of learning the basics of the language and the conventions for embedded development using this platform, we found that we were able to achieve multiprocessing using the new system to a degree that would allow us to finish the powertrain control portion of the software. We were able to implement throttle by wire as described in ESF2 without blocking the main execution loop, with the additional benefits of guaranteed memory safety provided by the language itself. This means that we have higher trust in our implementation, as software hard faults are almost entirely prevented at runtime through detection at compile time. This also opens the door to asynchronous communication with the electric motor controller over CAN, and potentially with other subsystems such as the AMS and the instrument panel.

In the future, we want to expand our software platform to include these additional subsystems. We also want to explore the possibility of reviving the MP5 codebase and building on top of that.