

Retrospective Sprint III

Team 04



Sprint III Goal

Establish the foundational communication link with the physical speedometer (**Input Capture**), achieve functional asynchronous data processing using **ThreadX** Real Time Operating System (RTOS), integrate full **Trustable Software Framework** with assertions example and ensuring readiness for OS-level integration.

PiRacer Battery Research

After thorough research and testing, we came to the conclusion that it's best to provide power to 2 different blocks: one for the brain, and another for the movement. We have prepared our current build for our future needs: a second pack of 3x18650 batteries with a BMS, fuses, buck converters and capacitors.

Component	Typical power	Peak power
Raspberry Pi 5 (16GB)	9–12 W	14–16 W
Hailo AI HAT	2.5 W	3 W
Seeed Dual-CAN HAT	0.45–0.6 W	0.6 W
USB SSD (NVMe via USB3)	5 W	7 W
Waveshare 7.9" touchscreen	3 W	3.5 W
TOTAL (Block 1)	20.95–23.1 W	28.1–30.1 W

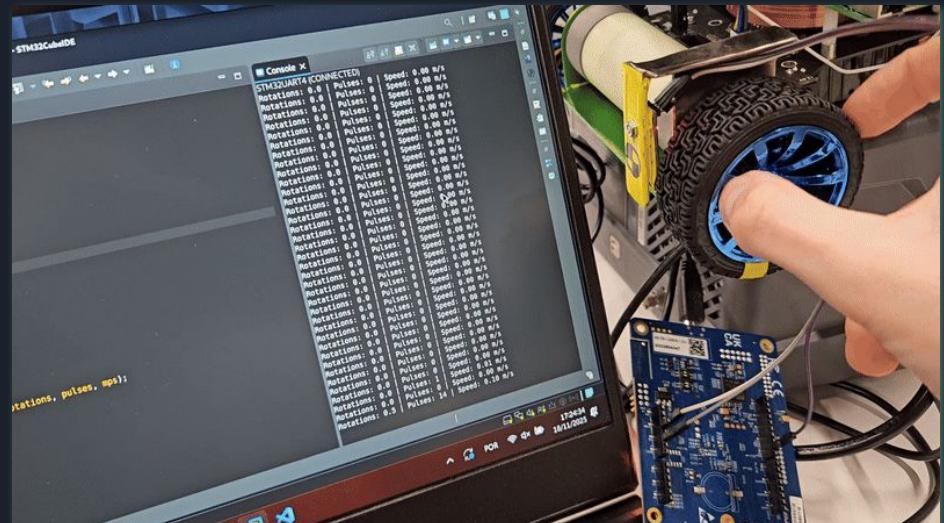
Component	Typical power	Peak power
STM32U585I-IT01A	0.5 W	0.5 W
LM393 speed sensor	0.01 W	0.01 W
CAN transceiver	0.2 W	0.2 W
MG996R servo (6 V)	6 W	15 W
PiRacer 37-520 motor #1 (6 V)	6 W	12 W
PiRacer 37-520 motor #2 (6 V)	6 W	12 W
TOTAL (Block 2)	18.7 W	39.7 W

STM32 & LM393 Speedometer

As the wheel spins, our custom 3D-printed encoder sends square waves of high and low voltage signals through the LM393.

The C code running in the STM32 successfully receives and parses these signals, converting them into m/s, using the real values.

It's timer-based (external TIM1), making it independent from other threads. Hardware filter makes sure signals are the purest.



$$Speed(m/s) = \frac{Pulses \times Perimeter}{PulsesPerRev \times Time}$$

STM32 & Wheel Control

As a last-minute breakthrough, we also successfully controlled the servo motors and DC motors through our C application running on the STM32.

The C code detects the Hardware ID of the components and wakes them up, delivering power to those specific components one by one.



ThreadX RTOS



Conducted a study of the ThreadX RTOS to understand its core concepts, scheduling model, and applicability to the PiRacer architecture.

Prepared and delivered a presentation and presented the findings to the community.

Analyzed and discussed the PiRacer's current software architecture and identify potential improvements.

ThreadX RTOS



Created and initialized the **Data Processing Thread**. It will wait to receive **speed data** from the **ThreadX Queue**, then process, log, or display it.

Initialized and configured the Eclipse ThreadX RTOS environment within the STM32 project: configuring the **system tick**, **memory allocation**, and preparing for **thread creation**.

Completed **ThreadX architecture** implementation with **queues**, **flags**, **mutexes** and **threads**, integrating **CAN communication** to transmit/receive operations.

AGL Integration And Qt Cross-Compilation

The **Qt Cross-Compilation** Toolchain on the host machine is correctly **configured** and **functional**. We can create executables **compatible** with the **Automotive Grade Linux** environment running on the target (Raspberry Pi 5 on SD).

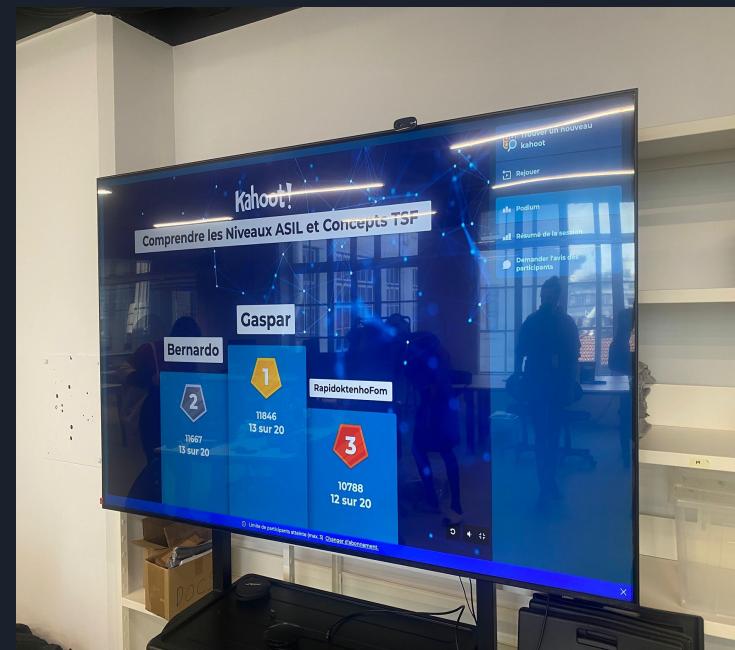
Although we couldn't fully install the system on the SSD, there were significant improvements on the configuration/preparation and our IC integration.



Trustable Software Framework And Compliance

Completed our second **Hands-on Lab for TSF** (Trustable Software Framework), learning critical skills for ISO 26262 compliance: **HARA** documentation, **ASIL** justification, **evidence** linking, and **SME** scoring methodology. In the end, a quiz was completed so the team could apply what they had learned.

- HARA (Hazard Analysis and Risk Assessment) creation
- ASIL calculation (Severity × Exposure × Controllability)
- ASIL justification writing
- Evidence artifact linking (code, tests, analysis, design, compliance docs)
- SME scoring methodology (score at LLTC level, propagates upward)
- Requirement chain assessment (URD→SRD→SWD→LLTC)



Trustable Software Framework And Compliance



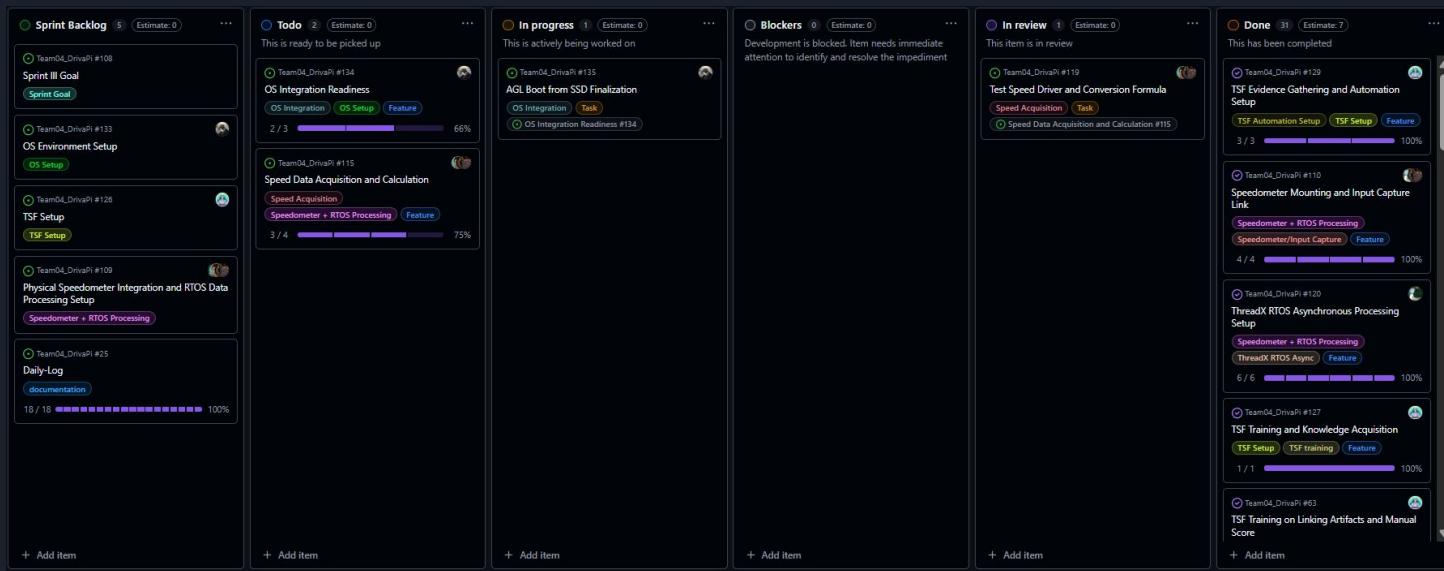
Integrated **GitHub Actions** to **automate** the process of building C/C++ projects, running **tests**, calculating code **coverage**, and performing static **analysis**.

Implemented **CI/CD** pipeline designed to enforce Trustable Software Foundation (TSF) validation on code changes. Its primary function is to **build** the code, **run verification** steps, **analyze** the resulting artifacts against **TSF rules**, calculate **trust scores**, and **report** the results back to the developer..

All of this is accomplished with **Custom Validators**, reading and parsing the verification artifacts to quantify their results into a **Trust Score**.

GitHub

We also changed the visibility of our repository to **public** to implement a **mandatory 2-Reviewer Rule** for **all Pull Requests** (to ensure code oversight). Additionally, we completed the all the tasks with the exception of the **AGL SSD Boot**. The code from the speedometer is still in review, but fully functional.



The GitHub Kanban board displays the project's workflow across five columns:

- Sprint Backlog**: Contains items like "Sprint II Goal" and "OS Environment Setup".
- Todo**: Contains items like "OS Integration Readiness".
- In progress**: Contains items like "AGL Boot from SSD Finalization" and "Speed Data Acquisition and Calculation".
- In review**: Contains items like "Test Speed Driver and Conversion Formula".
- Done**: Contains items like "TSF Evidence Gathering and Automation Setup" and "Speedometer Mounting and Input Capture Link".

Each card includes a progress bar indicating completion status. The "In progress" column shows the most active work items, while the "Done" column shows the completed tasks.

Thank you!

