NICOLA DI GRUTTOLA GIARDINO, FULVIO CASTELLO & NICOLÒ CARPENTIERI

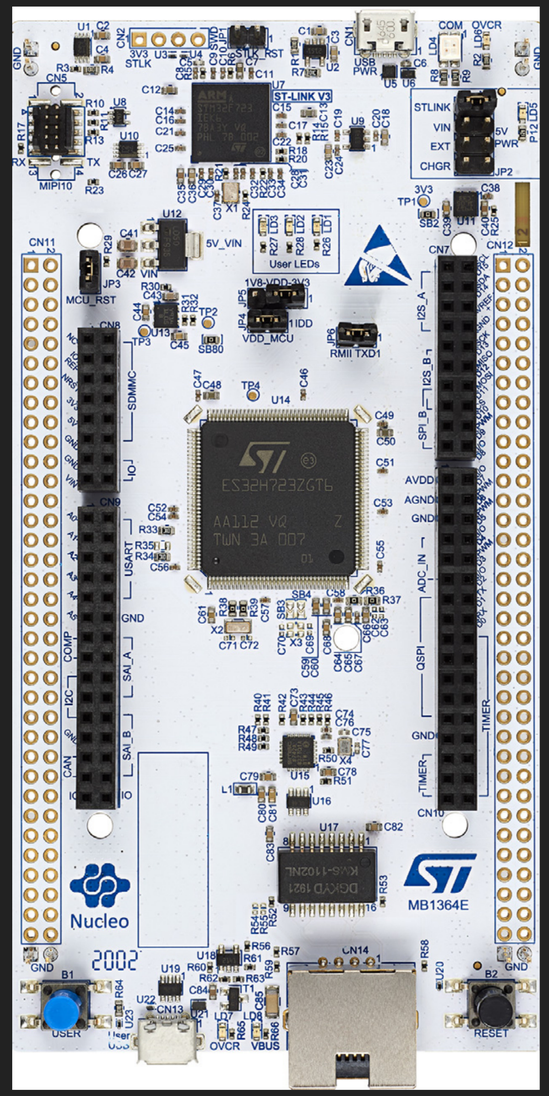
H7 STRESS TEST ⚡

INTRODUCTION ❖

- Main purpose: stress testing the scheduling capabilities of the CPU packed into a chosen embedded microcontroller
- Data processing: analog peripherals provide the input values necessary for the selected arithmetic computation
- Data storage: results can be either asynchronously read from or saved into an external memory, causing unwanted unpredictability
- Platform: RT-Thread Studio project based on the official kernel (v4.0.3) for the target OS, using the C standard programming language.

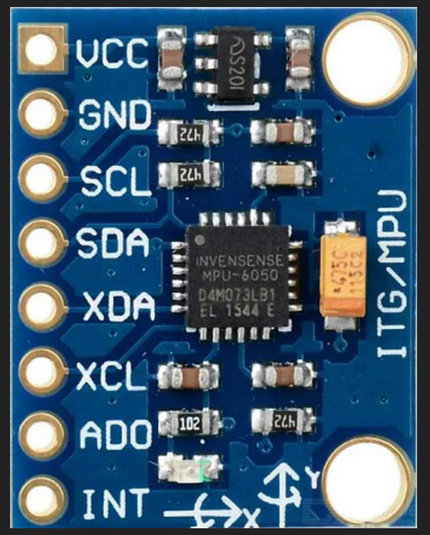
INTERNAL HARDWARE ARCHITECTURE ⌘

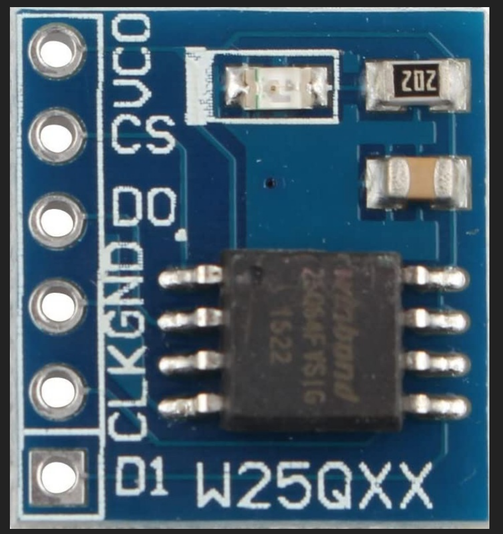
- STM32 Nucleo-144 development board equipped with an STM32H743ZI MCU:
  - 32-bit Arm Cortex-M7 core @ 480 MHz max.
  - Double-precision FPU
  - 32 Kbytes of L1 cache
  - 1 Mbyte of internal SRAM
  - I2C master(s) @ up to 400 kbit/s
  - SPI master(s) @ up to 150 Mbits/s.

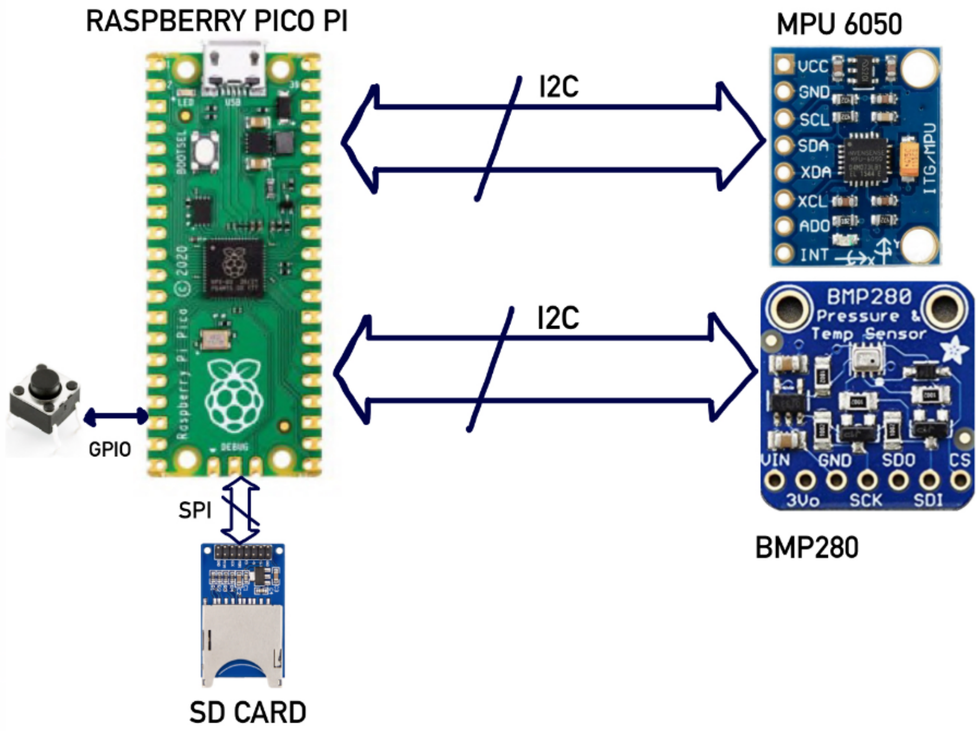


EXTERNAL PERIPHERALS ⏲

- BMP280 barometer providing ambient pressure readings via I2C
- MPU-6050 accelerometer feeding instant acceleration and gyroscopic data through I2C
- W25Q64FV 64 Mbit Serial Flash memory for results storage capabilities via SPI
- Two pushbuttons aimed at complementary asynchronous functionalities, both connected through a GPIO port.







CRITICALITIES ☒

- Hardware setup changes:
  - Raspberry Pico Pi, lacking support for console output functions
  - SD Card reader, missing file system implementation
  - Single push button, incapable of managing two distinct services in a unique package.

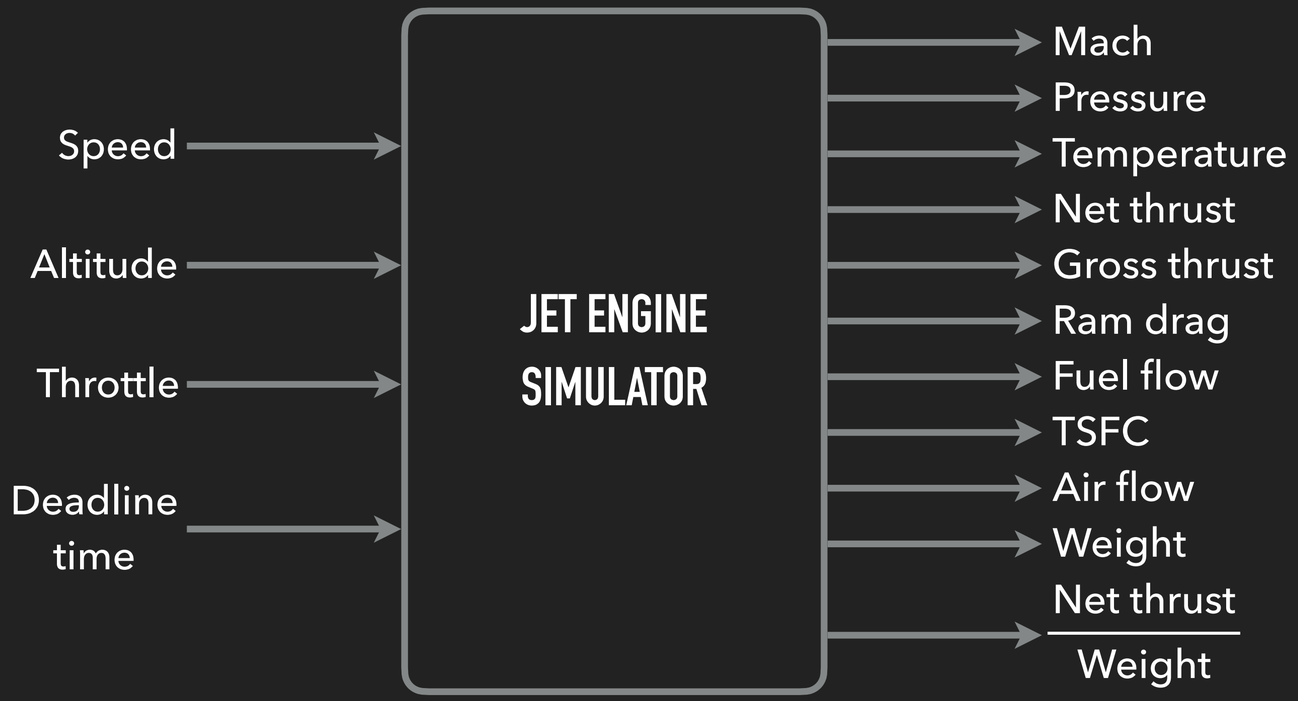
IMPLEMENTATION DETAILS: COLLECTING ANALOG DATA ✍

- Sequential reading of raw values from accelerometer & barometer in a continuously alternating fashion, with the highest priority overall:
  - MPU-6050 – after an initial peripheral reset, acceleration and gyroscopic data are read 6 bytes at a time, so that the final quantities can be calculated as the square norm of {x, y, z} components
  - BMP280 – following the required sensor configuration, eight 1-byte raw pressure values are gathered to then be transformed in bulk by means of a given conversion function
- All input parameters are stored inside dedicated arrays, which are protected from concurrent access by appropriate mutexes
- Each of them gets released by the corresponding thread as soon as enough values get successfully produced in order for the consequent benchmark to be able to generate the deriving results.

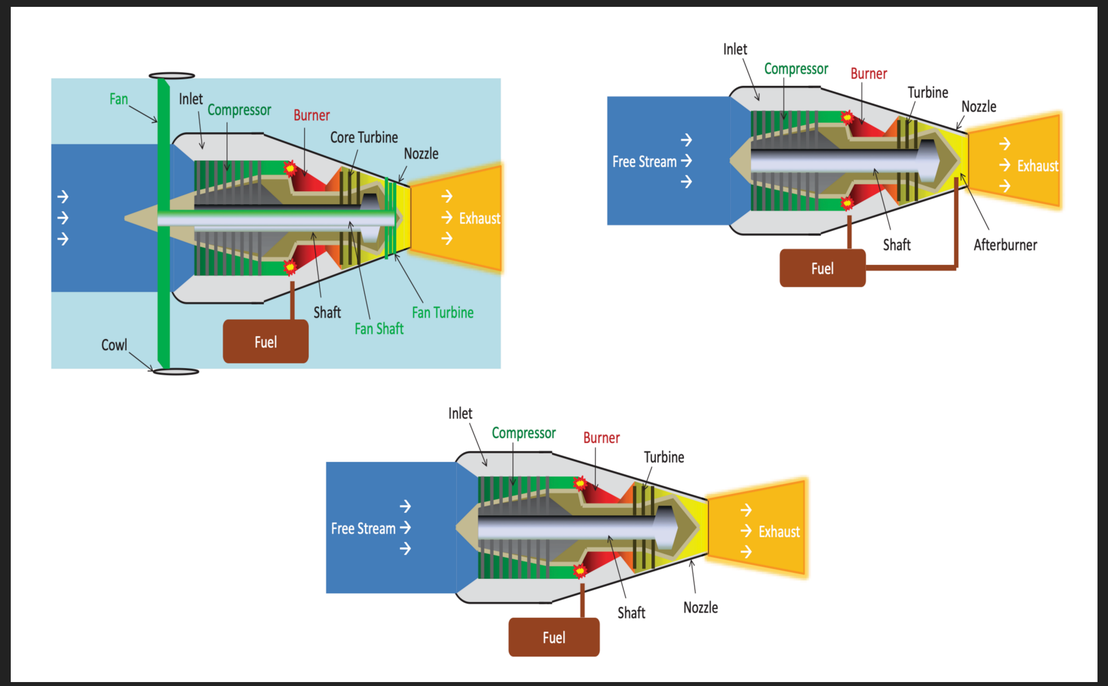
IMPLEMENTATION DETAILS: FLOATING POINT ARITHMETIC

- XenoJetBench was initially conceived as a multiprocessor benchmark based on thread parallelization offered by the OpenMP library and on hard-real-time support provided by the Xenomai framework
- For our purpose, it was necessary to strip down both the multi-core aspect and the internal deadline checks, as to reduce it to a much more trivial monolithic FPU stress test
- It's controlled by a main thread (with a lower priority compared to the producer) that consumes the input arrays deriving from the analog sensors by generating four intermediate-priority internal tasks
- Its total duration heavily relies on the chosen engine to simulate – automatic & random selection causes unpredictability: the worst case always needs to be taken into account
- Output data would represent various aerodynamic and structural parameters of the picked nozzle model, but they are of no interest in a context of just pushing the CPU to its limits
- They further lose meaning here, as our inputs don't match with their originally intended definition and are no longer read from a properly structured local file.

XENOJETBENCH: AN OVERVIEW ✈



- **Black-box** representation of the original entity, showing the former meaning of all the theoretical input and output variables.



- Available engine models: **Turbojet** (upper left corner), **Jet with Afterburner** (upper right corner) and **Turbofan** (down at the center).

IMPLEMENTATION DETAILS: SAVING SENSORS DATA & COMPUTATION RESULTS

- Whenever one of the pushbuttons gets activated, the associated thread tries to perform a write operation of either the benchmark inputs or its outputs into the first eligible address of the memory
- The chip is virtually split into two equal portions, respectively assigned to the data deriving from the producing thread and its consuming counterpart
- If there's no free space left inside the correct memory region, the entire memory simply gets erased, causing both pointers to restart from their initial values
- The actual writing sequence gets handled by low-level I/O functions, which were already present inside the RT-Thread kernel for the SPI protocol, through the use of messages
- Priority is the same as XenoJetBench main thread, causing a round-robin alternation to take place between them (depending on the status of the requested mutex).

IMPLEMENTATION DETAILS: READING BOTH GROUPS OF STORED VALUES

- The complementary thread has to perform a read operation from the most recently used address of the flash memory inside the suitable region
- This functionality trivially alternates with the previous one by means of a control variable that acts as a boolean value, inverting itself after each press of the corresponding pushbutton
- Low-level reading is again handled by means of messages, which get processed by the same internal SPI methods as before
- The newly acquired data gets saved into a temporary buffer and then processed into an appropriate variable to be printed out on the terminal console
- Priority and mutex handling are consistent, and as such multiple pushbutton presses get sequentially executed along with the actual benchmark in an alternating way.

REAL-TIME CONSTRAINTS ⌖

- Both aperiodic requests might take over iff the needed mutex is free, eventually lengthening the ongoing cycle and causing unacceptable delays
- Producer: releases its mutex after each reading sequence, has to finish before the benchmark starts
- Consumer: only retains its mutex while parsing input variables or saving final results, needs to complete its execution before the new round of analog data gets scanned
- If the interface between the two main threads gets violated, XenoJetBench internal variables end up updating mid-calculation as well, rendering its outputs unstable and causing synchronization issues.

IMPLEMENTATION DETAILS: DEFERRABLE SERVER ⌚

- A huge leap forward compared to the more basic asynchronous approach implemented within RT-Thread by default (so far…)
- Defined as just a more complex thread by using standard kernel libraries, it includes some additional features aimed at handling aperiodic tasks
- Capacity and period are implemented by means of two dedicated timers, which suspend the server itself whenever either of them runs out
- A list of aperiodic requests is included to manage them by a “first come, first served” rule – always subject to the overall priority of the server.

IMPROVEMENTS ☑

- Introduction of the deferrable server algorithm:
  - Asynchronous requests conveniently added to its queue right at their arrival time
  - Total aperiodic execution time strictly controlled by its capacity and period, so system failures never occur
  - Allows for a shorter macro period by pushing redundant IRQs to the following cycle(s).
- Fixed parameters imply better predictability:
  - Period & capacity large enough to accomodate both aperiodic operations within one cycle (accounting for the worst case)
  - Priority equal to XenoJetBench, inducing a round-robin rotation between them iff the required mutex isn't taken.

SYSTEM BENCHMARKING & FINAL COMPARISONS ⚙

Configuration ☞	Basic [s]	Enhanced [s]
t_{idle} – total	245	206
t_{idle} – benchmark	45	6
$t_{arrival}$ – benchmark	5	5
t_{idle} – sampling	240	201
$C_{aperiodic}$ – server	∅	1
t_{idle} – server	∅	205

“AN RT-THREAD-BASED H7 STRESS TEST, PROCESSING RAW SPATIAL AND AMBIENT DATA TO MAXIMIZE CPU UTILIZATION ON AN STM32 EMBEDDED PLATFORM.”

OSES Project