Universidade de Aveiro

DETI

Report for the third practical project

Prototyping a thermal process control system using the nRF52840-DK and Zephyr

Course of SETR at UAveiro

June of 2025

Teacher:

Professor Paulo Pedreiras

Autors:

Pedro Ramos
p.ramos@ua.pt
n.º 107348
Masters in Robotics
and Intelligent Systems

Rafael Morgado
rafa.morgado@ua.pt
n.º 104277
Masters in Robotics
and Intelligent Systems

First Project

1. Introduction

This project focuses on the development of a prototype of a thermal process control system, which is composed of an heating element and a temperature sensor. The controller used is the Nordic nRF52810-DK and communications with the system are done via UART (for configuration) or using physical buttons (for normal operation).

The real-time operating system Zephyr was used in this project, since it provides a low-level programming environment (using C) and comprehensive hardware abstractions for manipulating the board and other components.

Additionally, this project continues the implementation of unit tests on our codebase via the Unity test tool, which helps us to ensure the accuracy and reliability of the codebase. The CMake, Doxygen and Github project tools have also been utilized to improve the overall quality of the implementation.

2. Project links

Github: https://github.com/P-Ramos16/SETR_Proj3

3. How to build and execute the project

Build and Flash the Main Project

In VSCode, load the prj.conf located at the root of the project. Built the project and flash it to the controller.

Run the Unit tests

```
$ cd $PROJECT_ROOT/tests/src
$ cmake ..
$ make
$ ./rtdb_tests
$ ./cmdproc_tests
$ ./PID_tests
```

Generate and Read the Documentation

```
$ cd $PROJECT_ROOT
$ doxygen Doxyfile
$ firefox docs/html/index.html
```

Build the Hardware

The temperature sensor, FET and heater element (here represented by a blue LED, for simplicity).

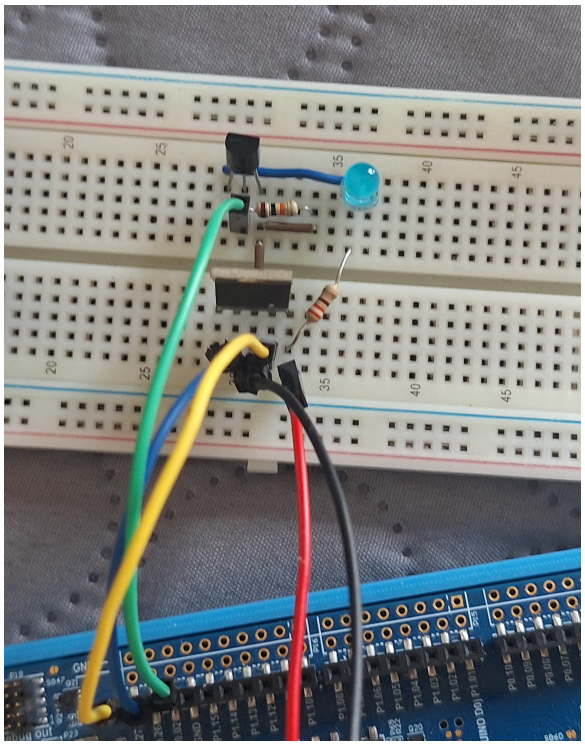

Figura 1: Top view of the hardware.

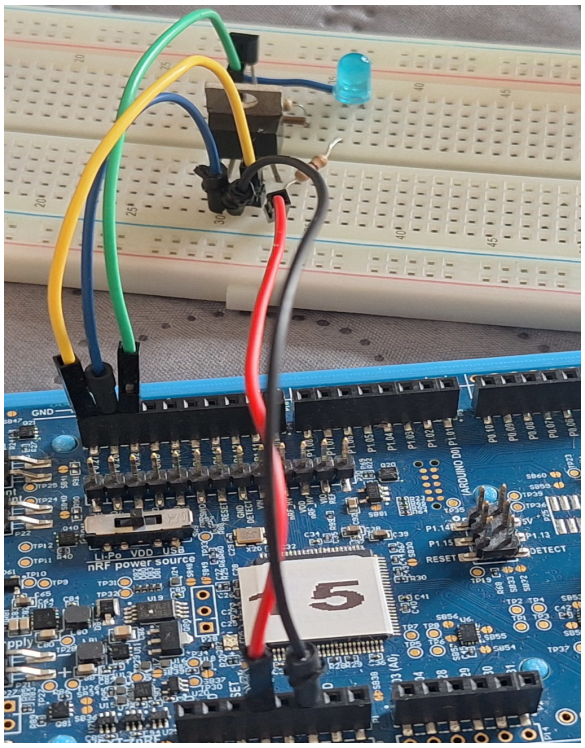

Figura 2: Pinout view of the hardware.

A 10kΩ resistor was used between the Gate and Source pins of the FET. A 220Ω resistor was placed in series with the blue LED. The pin number 27 and 26 were used for the I2C communication with the temperature sensor and the pin number 2 was used for the gate of the FET. Power was given through the VDD pin and a generic GND pin. The FET is connected to the negative side of the LED.

4. Tasks

- **Button Interrupt**: Updates the desired temperature and system On/Off status via interrupts;
- **LED Update Task**: Updates the LED indicators based on the system state;
- **Temperature Reading Task**: Reads the current temperature from the sensor and updates it on the rtdb;
- **PID Controller Task**: Calculates the On/Off status of the heater based on the current temperature using a PID controller;
- **Heat Control Task**: Controls the heater activation based on the decision of the PID controller;
- **UART Command Interrupt**: Processes incoming UART signals until a full command starting in '#' and ending in '!' is sent. It also handles the TX/RX UART buffers and UART transmission;
- **Command Processor Task**: Processes received UART message and responds accordingly.

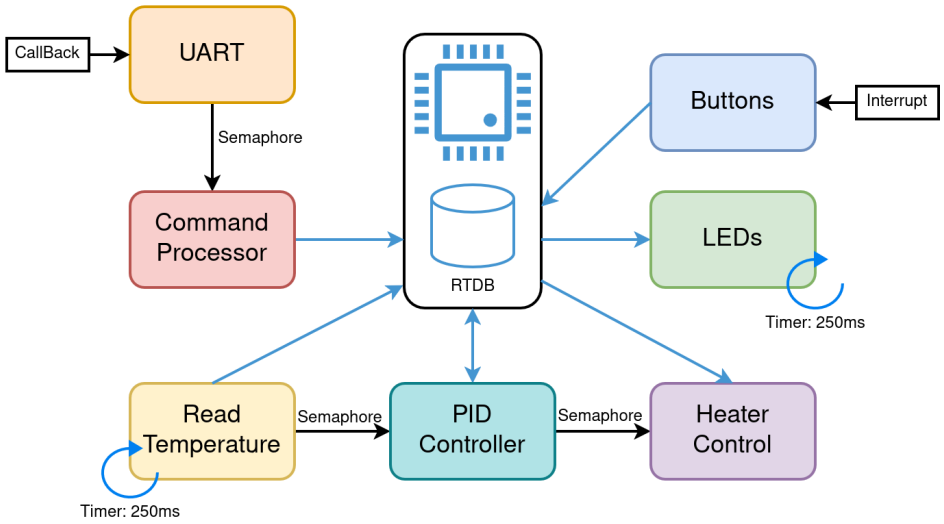

Figura 3: Task architecture.

Each task waits on either a timer or an event (semaphores or hardware callbacks) to be executed.

The board's buttons are handled by a callback function to the hardware interrupts, so it is event-triggered by a button press. This task does not depend on any other and does not require a timer. It updates the rtdb whenever a button is pressed. This task does not indirectly call any other task, as it doesn't need to influence the general workflow.

The LEDs are a separate task that are not "mission critical". The task changes the status of each LED according to the last stored values on the rtdb and does not depend on any other task. Since it is the lowest importance task, it is executed every 250 milliseconds. This is a moderately long time but is quick enough for a human to perceive well the status of the program.

The temperature read task also executes every 250 milliseconds and reads the temperature value from the I2C sensor and stores it in the rtdb. The 250 milliseconds where chosen due to the refresh rate of the I2C sensor and the time it takes for the heating element to influence the physical temperature around the sensor. After this task is executed and the rtdb value is updated, a semaphore that triggers the PID controller is lifted. The task is time-triggered since we have to ask for a reading from the sensor, the sensor itself does not say when a new value is available, so it cannot be event-triggered.

When a new temperature reading is available, the PID controller receives a semaphore "up" signal that allows it to execute. The PID controller simply takes the new value and the intended temperature value and decides if the heater element should be On or Off. This decision is also stored on the rtdb. At the end, a semaphore that triggers the heater control is lifted. The task is event-triggered since it a PID controller keeps track of the value history, so it only makes sense for it to calculate the output whenever a new value is received, or else it would continuously utilize the old value and also make the historical part of the control inaccurate.

The heater control simply reads the On/Off value for the heater from the rtdb and turns on the FET gate via a GPIO pin. It is event-triggered, since it should only operate when a new decision from the PID controller is available, as the heater state is not altered by any other task.

The UART reader iterates over each character received from the USB interface and fills a buffer with the new message received. The buffers only start filling when the "#"

starting character is received and stop when the "!" end character is received. This task works via zephyr's UART callbacks and opens a semaphore for the command processor to start working with the received message. This task also takes care of the input and output UART hardware buffers. This task operates after a "character received" event occurs.

The command processor verifies new messages and processes them according to the data present in them. It works similarly to the one in the second project and retains most of its aspects, only changing the commands accepted and the responses. One difference is that now all commands are acknowledged, so the command processor now generates responses for every single possible input message, independent of its valid or not. The outgoing messages are sent to the UART task's transmission buffer, so they are not sent by the command processor itself. The task only operates whenever a new complete UART message is received, since it does not make sense to continuously check the buffers for that, and might lead to memory conflicts with the main UART task.

5. RTDB

The rtdb is a database with a shared interface that enables values to be stored and shared among the different tasks.

Accesses to shared data are protected by mutexes that allow only one task to access each resource at a time, meaning that all other tasks need to wait for the initial task to finish. Each variable has its own mutex (except the PID parameters, which are grouped) which is locked by the first task entering the "dangerous" part of the code. It is then unlocked when the task finishes. This ensures that values are not updated while another task is utilizing them or overwritten while a task is reading them. The mutexes stay locked forever until the task is over with the data manipulation process. The decision to have an independent mutex for each variable allows multiple threads to operate at the same time on the rtdb as long as the accessed values do not interfere with each other.

6. System Schedulability

The system's schedulability is ensured by strictly controlling the execution of each task to guarantee that no calculations are unnecessarily (and wrongly) performed on stale data. The limited use of time-triggered tasks and the preference for event-triggers helps maintain the real-time responsiveness of the most critical jobs.

- **Button Interrupt**: Event-triggered by hardware interrupts, this task has the highest priority to ensure immediate response to user inputs. Its execution time is very small, as it performs minimal processing;
  - *Behavior*: Updates RTDB (minimal processing).
  - *WCET*: Less than 1 ms.
- **LED Update Task**: Executes every 250 ms. The interval is sufficient for human perception of changes in the LED;
  - *Behavior*: Updates LEDs based on RTDB state.
  - *Rationale*: Low-priority, human-perceptible interval.
- **Temperature Reading Task**: Time triggered every 250 ms and starts the "waterfall" of secondary tasks that take care of the data processing and output;
  - *Behavior*: Reads I2C sensor, updates RTDB, signals PID.
  - *WCET*: 10 ms (I2C overhead).

- **PID Controller Task**: Event-triggered by the temperature reading task, as there is no reason to work on values that have not been updated yet and may cause the PID calculations to be wrong;
  - *Behavior*: Computes heater state (On/Off).
  - *WCET*: 4 ms (floating-point arithmetic).
- **Heat Control Task**: Also event-triggered by the PID Controller, since it only needs to execute after the PID creates a new intended state for the heater;
  - *Behavior*: Adjusts heater via GPIO.
  - *WCET*: 0.5 ms.
- **UART Command Interrupt**: Event-triggered by UART character reception. It has some processing to minimally parse received characters to avoid calling the command processor on obviously incorrect inputs;
  - *Behavior*: Parses messages (start/end flags), signals Command Processor.
  - *WCET*: 3 ms/character.
- **Command Processor Task**: Event-triggered by the UART Command Task whenever a "correct enough" message gets received, as there is no point in processing incomplete messages.
  - *Behavior*: Validates/executes commands, queues responses.
  - *WCET*: 5 ms (worst-case parsing).

Worst-Case Execution Time: The system is designed such that the sum of WCETs for all tasks within their respective periods does not exceed the available CPU time. For example

- The interrupts are optimized to have as little computation as possible, as to not disrupt other tasks for very long due to the their high priority;
- The timer tasks are as lightweight as possible, with little computation, but the hardware computations can be on the heavier side (GPIO manipulation or I2C communication);
- Event-triggered tasks (PID, Heater Control) execute infrequently and briefly, ensuring they do not disrupt the timing of the periodic tasks.

7. Additional features

When the "V" command is sent via UART, the program goes into "verbose" mode and outputs what every task is doing and a timer (for some) to show if they are getting backed up or are executing on time.

The "D" command allows the user to request the currently desired temperature, helping with debugging and checking the current system status.