TemperatureMonitor Project Documentation

Contents

1 Overview

The TemperatureMonitor project is designed to monitor temperature values, control LEDs based on thresholds, and interact with hardware interfaces such as sensors and actuators. This documentation provides an overview of the project structure, main components, and usage instructions.

2 Design principles

The project's goal is to show an exemplary architecture for an implementation of a temperature monitoring system in an embedded environment. The design was made under the following constraints:

- Modularity
- Separation of Concerns
- Encapsulation
- Form follows function

In the following, each aspect is explained in more detail, and how it is implemented in the code.

2.1 Modularity

2.1.1 C Version

The C-Code is structured in terms of individual files with respective header files that define the structure. The interfaces are defined in terms of structures. The advantage is that the function calls can easily be extended without having the need to rewrite the whole code.

2.1.2 C++ Version

The C++-Code is also structured in terms of individual files but also in terms of different class definitions. Moreover, this includes the spectrum of template classes to allow for maximal reusability and extendability.

2.2 Separation of Concerns

2.2.1 C Version

The C-Code is structured in layers in such a way that the hardware is abstracted in terms of signal types, e.g., raw signal is a uint16_t coming unscaled from the ADC conversion and then lifted to a int16_t which is scaled correctly to a physically meaningful type.

Also, all hardware related functions are implemented respecting timing issues. Reading the EEPROM is made only once at the beginning of the program - as there was no specification if the EEPROM shall be made hotswappable.

The update of the ADC temperature sensor is mocked inside a separate task that is called whenever a key is hit. That reflects the behavior as it would be in an embedded system.

All values are stored in global variables but only accessed through a functional interface. No direct access to the sensors or actuators is permitted in order to keep the abstraction.

2.2.2 C++ Version

In the C++ version, there is also a distinction between the different level of abstraction layers, coming from the hardware level up to the application level.

In contrast to the C-Code, the C++ version employs many advantages of the object oriented thinking. Every class admits a base class that defines the required interfaces. So, every sensor-like hardware must be derived from the sensor class which implements a `updateRawValue`

function in order to assure that the update of a time-critical function, as it is the case for an ISR call, is performed correctly.

An actuator, on the other hand, employs a `read` and a `write` function and is therefore derived also from a `sensor` class. This is motivated by a similar idea as in the ros2_control architecture.

2.3 Encapsulation

2.3.1 C Version

Standard C Code does not allow for encapsulation as it is known from an object-oriented language. However, using `structs` for interfaces and separate files for each architecture unit brings it close. This was employed here and is explained in the previous paragraphs.

2.3.2 C++ Version

As mentioned before, all units of the project are written in classes in a sensible manner to allow for a simple, yet powerful and extensible structure.

2.4 Form follows Function

Dealing with embedded systems means that there are strong timing and resource constraints. To ensure a high flexibility while satisfying these constraints, it is important to provide an architecture that does this naturally.

2.4.1 C++ Version

The classes are there defined such that their respective methods are self-explained and each one allowing for a specific level of abstraction. So does the `sensor` class directly implement a function that is meant to be called inside an interrupt handler and performs a minimal function to allow for a quick execution.

3 Outlook to further improvements

- Implement unit tests for each module
- Add more detailed error handling and logging mechanisms
- Use a real-time operating system (RTOS) for better task management
- Use a device-tree like structure to define the hardware interfaces (see e.g. here for more information or Zephyr Project OS documentation).
- Use names or something equivalent to identify the different hardware interfaces instead of hardcoding them in terms of numbers.
- Implement a more sophisticated simulation environment for testing

4 Features

- Periodic temperature monitoring
- LED control based on temperature thresholds

- Keyboard interaction for simulation
- Modular hardware abstraction
- Simulated EEPROM access

5 Getting Started

5.1 Requirements

- GCC or compatible C compiler
- G++ or compatible C++ compiler (for cpp)
- Make
- (Optional) VS Code for development
- pandoc and pdfcrop for building the documentation

5.2 Build Instructions

C Version

```
cd c
make
```

C++ Version

```
cd cpp
make
```

Documentation

```
cd doc
make
```

5.3 Running the Application

C Version

```
make run
```

C++ Version

```
make run
```

6 Code Structure

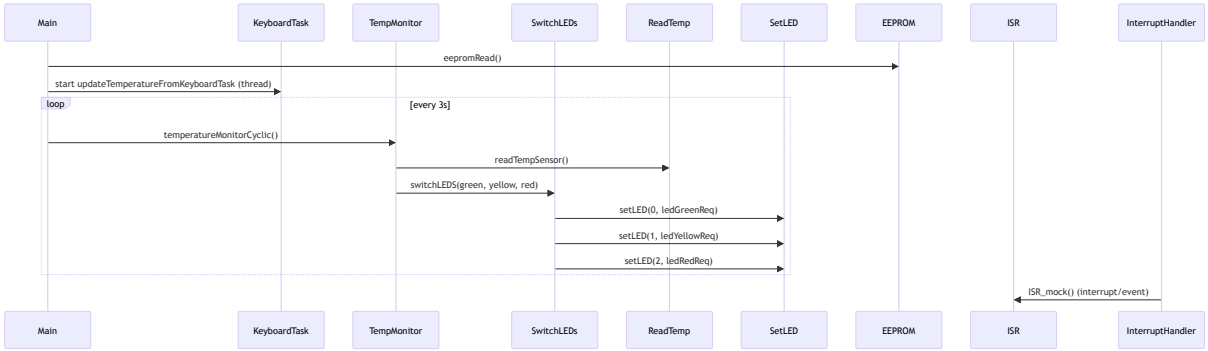
6.1 C Code Call Sequence

The idea behing the implementation is to mock a cyclic task that is usually found in embedded systems. This cyclic task is responsible for updating the temperature value and controlling the LEDs accordingly. The keyboard input is handled in a non-blocking way to simulate interrupt-driven behavior.

The main logic is in the cyclic task implemente in `temperatureMonitor.c`. It reads the temperature from the sensor, checks against thresholds, and updates the LED states. It is important to mention, that the sensor readigns and actuator writings are based on function calls to the hardware abstraction layer defined in `hardwareInterfaces.c`. This is an important design choice to allow for a modular and easily extendable architecture.

When implementing another application, the interfaces to the hardware may stay the same. Due to the functional interface, the priorization can either be done in the hardware abstraction layer or in the application layer.

The prototypic call sequence for the temperatureMonitor is depicted in 6.1.



6.2 C++ Code UML Chart

In contrast to the C-Code, the C++-Code is structured in classes. Each class has a well-defined purpose and interface. The main classes are depicted in 6.1.

The most basic class is an (almost) abstract template class for the sensors. The `updateRawValue` method is meant to be called inside an interrupt handler to ensure a quick update of the raw value. It is implemented as final to assure that no derived class can override it and potentially introduce timing issues. The `read` method is virtual and meant to be called from the application layer to read the processed value.

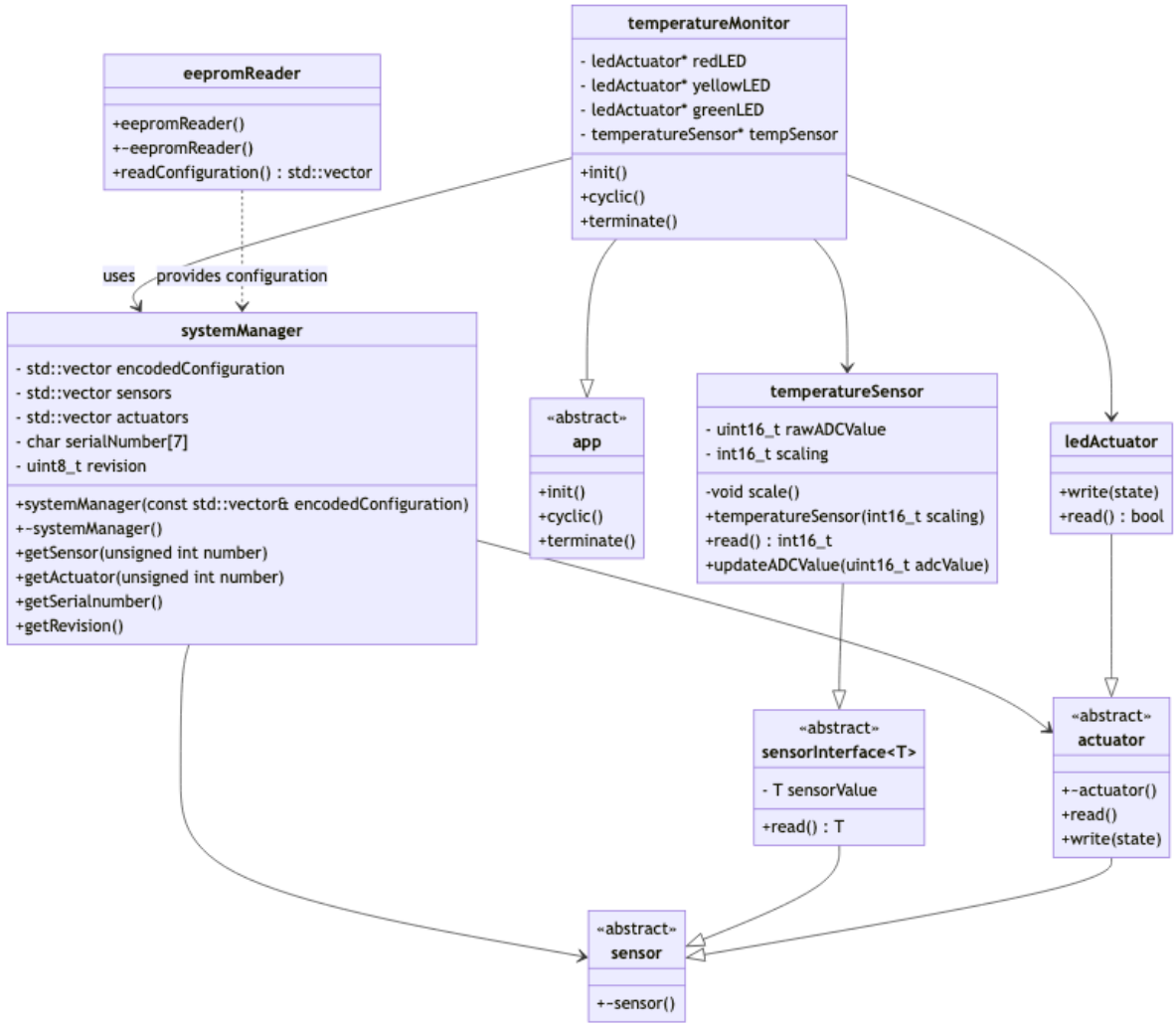
Another important class is the actuator class that is derived from the sensor class. It implements a `write` method to set a value to the actuator. It is derived from the sensor class in order to be able to reflect internal states of the actuator which potentially be reead using an interrupt handler. In the example however, the internal states are directly copied from the written value. As mentioned before, this is inspired by the ros2_control architecture.

Another important abstract class is the `app` class. It contains three abstract methods: `init`, `cyclic` and `terminate` and are meant to be called by the main program handler before starting, during cyclic execution and at the end of the program, respectively. It is the base class of all application implementations, including the `temperatureMonitor` class.

The `systemManager` class is a singleton class that is responsible for managing the system state, including initialization and termination of the application. It provides the hardware interfaces to the application layer. This is done by defining the hardware interfaces as member variables and providing getter methods to access them. The definition is read using a

mocked EEPROM interface at the beginning of the program execution. To stay withiin the object-oriented paradigm, the EEPROM interface is also defined as a class `eepromReader`. The idea of the SystemManager is based on the *Factory design pattern*, see e.g. here for more information.

For a new application, one would derive a new class from the `application` class and implement the required methods. The hardware interfaces can be accessed using the `SystemManager` singleton instance.



7 Usage

- Use the keyboard to simulate temperature changes:
  - Press `w` to increase temperature.
  - Press `s` to decrease temperature.
- The LED states will update based on the current temperature value.

8 License

This project is licensed under the MIT License.