Embedded Systems and Internet-of-Things
-
Third Assignment

Kimi Osti

February 26, 2025

Contents

Chapter 1

System Requirements

The system is a smart IoT-based temperature monitor. In particular, it measures a closed environment's temperature at any given time, and controls a window connected to a motor to properly ventilate the room in case of critical temperatures. It also implements a manual mode, which can be activated via a button on the *Window Controller*, or via a web-based *Operator Dashboard*, that allows an operator to physically control the window opening angle thanks to a potentiometer attached to the *Window Controller*.

1.1 Temperature Monitor

It's the main system component. It periodically measures the room's temperature, and communicates it to the *Control Unit*, which is responsible of controlling the other component's behavior according to the valued registered by this subsystem. The frequency of the measurements depends on the state of the system, which is stored in the *Control Unit* and is communicated to this component in real-time. It's connected to the *Control Unit* via the *MQTT* protocol, and must include a LED signaling whether the connection is properly established.

1.2 Window Controller

It's the in-place operator interface. It's responsible of physically triggering the window movement, according to the values communicated by the *Control Unit* if the system is in automatic mode, or according to the value controlled by the potentiometer if the system is in manual mode. It has a button to switch between these two modes, and it also has a screen that tells the

operator the state of the system at any given time. All necessary info is communicated by the *Control Unit* via Serial communication.

1.3 Operator Dashboard

It's a web-based user interface that allows operators to work remotely on the system. It shows a graph representing the current state of the system and a brief history of the last measurements, sided by a statistic showing the average, minimum and maximum values in the last period of time. In addition to this info, it exposes a simple operator interface which allows the user to switch between manual and automatic mode, as well as to restore the system status in case an alarm was triggered. It communicates with the *Control Unit* via the *HTTP* protocol in order to reflect user actions on the actual in-place subsystems.

1.4 Control Unit

It's the core of the entire system, and it serves as a mediator between subsystem interactions. Its main function is to track the system state and all info related to the measurements and to the current operating mode (manual or automatic). It also determines the sampling frequency for the *Temperature Monitor* according to the current measure, and the window opening percentage according to the system state (if in automatic mode). It's also responsible of storing all system data, including the last measurements that the Dashboard shows in its graph, and the average, minimum and maximum values that are shown to the operator.

Chapter 2

System Architecture

This system can be structured dividing responsibilities according to the MVC architectural pattern.
In particular, the Controller behavior is implemented by the **Control Unit**, while the main View component interfacing towards the user would be the *Operator Dashboard*. The *Temperature Monitor* is fully a Model component, since it samples the real-world status value and shares it with the *Control Unit*. Finally, the *Window Controller* can be considered to be almost completely Model, with a part of View in the Control Panel, which allows the user to switch mode between manual and automatic, and to control the window angle when in manual mode.

2.1 Temperature Monitor

This subsystem's main feature is sampling the room's temperature with a given frequency. It's also responsible for sending that data to the *Control Unit*, so it must include a component handling the subsystem connectivity to the back-end.
Its detailed behavior can be described analyzing each sub-component with its responsibilities. Each sub-component can be modeled as a Task, and all Tasks run in parallel in the subsystem on which this part of the application is deployed. All Tasks can be modeled as Synchronous Finite State Machines.

2.1.1 Temperature Measuring Task

The central Task is the one responsible of actually measuring the temperature.

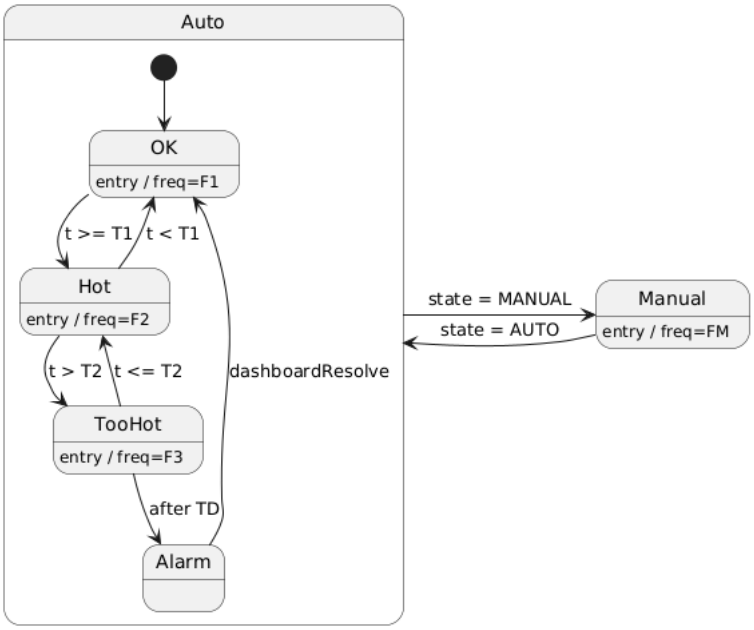

Figure 2.1: FSM modeling the temperature measuring task behavior

It is clearly modeled - for clarity - how the state transitions in automatic mode are determined by the temperature values. But in reality, all state transitions are demanded by the *Control Unit* according to the values that it receives from all components, since it is the central Controller for the application.
Also, it is clearly modeled to be network-agnostic. Indeed, this sub-component is meant to measure at all times, and it will be a responsibility of the network-related tasks to retrieved the stored data when it's supposed to be shared.

2.1.2 Connection Monitoring Task

This task is the one responsible for monitoring the system connection state. It's supposed to check whether the component is still online, and when it's not, to try and reconnect. It works on two layers: the first one checks whether the system is connected to the *WiFi*, and the higher one checks whether the *MQTT* subscription is still on.

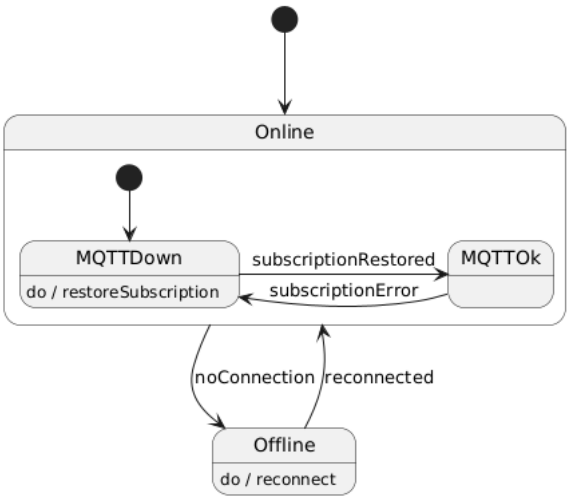

Figure 2.2: FSM modeling the connection monitoring task behavior

2.1.3 Communication Task

This task is responsible of communicating with the *Control Unit* via the *MQTT* connection set up by the *Connection Monitoring Task*. It ensures that data is properly collected, assembled to form a message and sent to the *Control Unit*. On the other hand, it's also responsible of receiving the response messages published by the *Control Unit*, which can dictate the *Temperature Measuring Task* state, and consequently its sampling frequency.

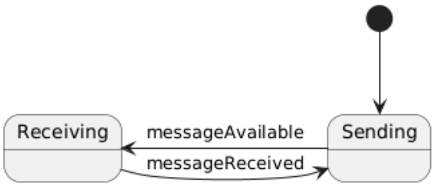

Figure 2.3: FSM modeling the communication task behavior

It's here clear how this task too is network agnostic: simply, when there is no connection or no active *MQTT* subscription no messages are received - and the system therefore never goes to the *Receiving* state - and the ones that are sent might be lost.

2.1.4 LED Task

This task's main responsibility is to represent to the user whether the sub-system is online.

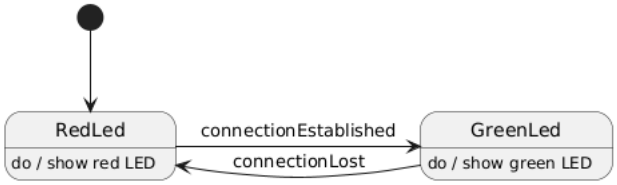

Figure 2.4: FSM modeling the LED task behavior

In this scheme, it's obvious how this Task's state depends directly on the state of the *Connection Monitoring Task*. In particular, the LED is shown as green only when the device is fully connected to the Internet and the *MQTT* subscription is running properly.

Note the sub-system's connection state is kept internally, and is not communicated to the *Control Unit*. This because, architecturally speaking, the rest of the system is not concerned about this specific sub-system's connection state, since it would be simply cut out of the communication network. So, connection state is here only tracked to help the LED Task show the user the correct information, but the rest of the system - if well decoupled - is required to keep running even if the communication went down (or theoretically speaking, also if there were multiple devices tracking the temperature and connected to the same *MQTT* topic and broker).

2.2 Window Controller

This component's main behavior is to actuate the window opening. In doing so, it's connected to the *Control Unit* via *Serial Line* and it exposes a little operator panel to directly manipulate the window, switching between manual or automatic mode and controlling the window opening level, if in manual mode. Its behavior can be modeled with multiple Tasks, each one represented by a Synchronous Finite State Machine.

2.2.1 Window Controlling Task

This Task is responsible of actuating the system's behavior on the actual window. Its only duty is to set the window state each period, querying the

Communication Task to retrieve the current opening value signaled by the *Control Unit*.

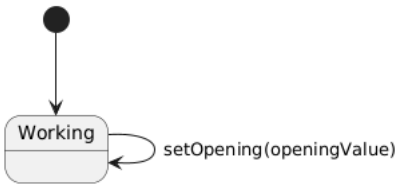

Figure 2.5: FSM modeling the window controlling task behavior

It's obvious how this task's behavior is agnostic of the state of the rest of the system. Simply, each period it retrieves the last received opening value (which is managed completely by the *Control Unit* for consistency reasons) and actuates it to the window via the servomotor.

2.2.2 Operator Input Task

This Task is responsible of collecting user input, and of communicating it to the *Control Unit* to modify the entire system's state accordingly.

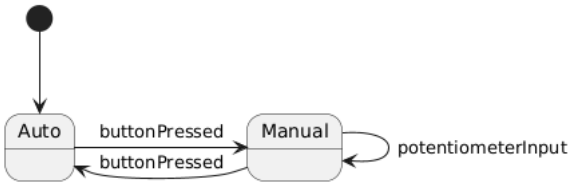

Figure 2.6: FSM modeling the operator input task behavior

In this scheme it's underlined how this tasks accepts operator input via the potentiometer only in manual mode. For better consistency though, data is here not directly sent to the *Window Controlling Task*, but rather a decoupling level is inserted between the two, centralizing state handling responsibilities in the *Control Unit*.

2.2.3 Operator Output Task

This task is responsible of showing the operator some information about the system's current state. Obviously, for the reasons described above, the system's state is described as received from the *Control Unit*, delegating it all data consistency issues.

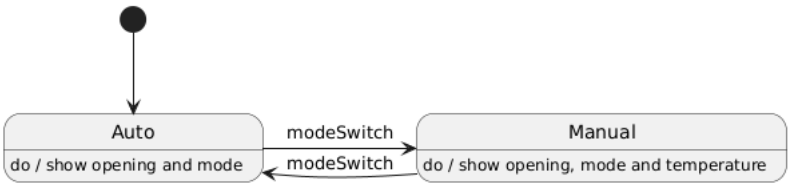

Figure 2.7: FSM modeling the Operator Output Task

In this scheme, it's depicted how this task is showing different information according to the state of the system. On the other hand, state transitions are not handled in any way by this task, and the *modeSwitch* is some input received by the *Control Unit*.

2.2.4 Communication Task

This task is responsible of communicating via the *Serial Line* with the *Control Unit*. It's a core feature of this sub-system, since correct communication ensures data (and more importantly behavior) consistency when translating computation into actuation.

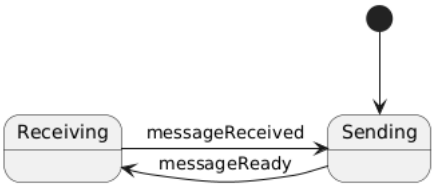

Figure 2.8: FSM modeling the communication task behavior

The Communication Task checks for new messages periodically, and expects to receive each period a message containing the current opening level, the current mode and the current temperature. On the other hand, it periodically sends a message containing a "dirty" state, which is the one requested by the user via the operating panel. It's then a *Control Unit*'s responsibility to properly modify the actual state of the system.

2.3 Operator Dashboard

The *Operator Dashboard* serves as a remote user interface to control the system. In particular it offers a graphical representation of the system's current state - which includes a graph of temperature history, the current average,

maximum and minimum temperature values in the last period of time, the system operating mode and the window opening level. In addition to this, it allows the operator to switch between automatic and manual modes, as well as a button to restore the system after solving an issue which caused an alarm. It's connected to the *Control Unit* via *HTTP*.

2.4 Control Unit

This sub-system is the core of the application. It exchanges data with all other sub-systems, and ensures data consistency storing internally all relevant values. Its main responsibility is to bridge between protocols to coordinate all system components, and to actuate the system's behavior on the real world.
This sub-system itself is made out of various components.

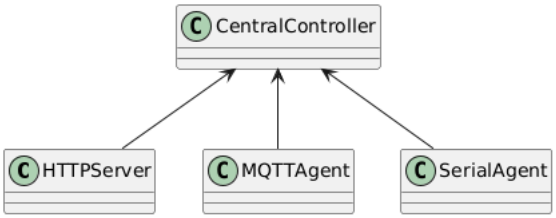

Figure 2.9: UML class diagram showing the architecture scheme for the Control Unit sub-components

In this scheme, the *CentralController* is the one responsible of ensuring proper data consistency, and it cooperates with all the other sub-components to properly communicate with each sub-system.
The main concern implementing this sub-system is to ensure that data is properly stored and can be safely accessed, preventing race conditions and other concurrency issues.

Chapter 3

Implementing Solutions

3.1 Temperature Monitor

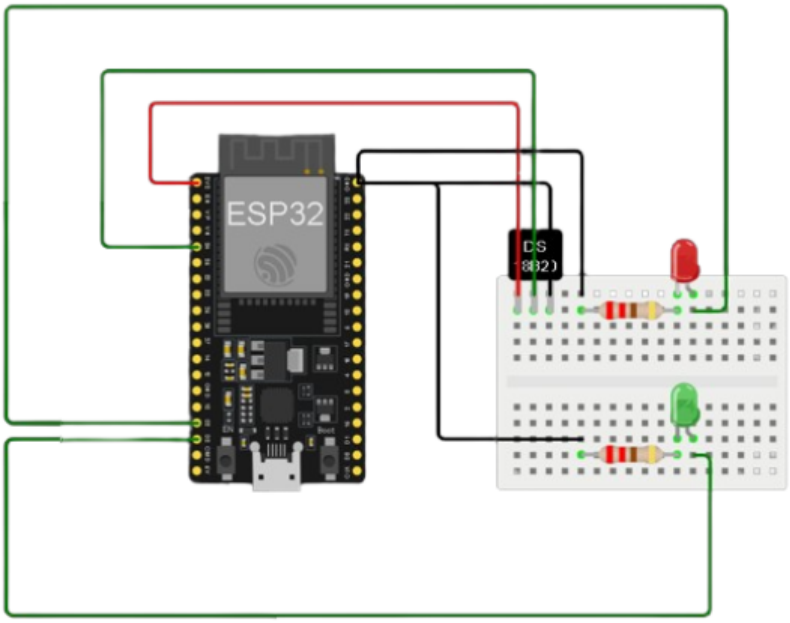

Figure 3.1: Detailed circuit for the temperature monitor sub-system

In this scheme si depicted the detailed circuit for the *Temperature Monitor*, deployed on an *ESP32 System-on-a-Chip* and programmed relying on the *Wiring* framework.
The *ESP32* holds enough program memory to host a little Operating System, which means that the programmer can deploy Tasks on both its cores

without having to directly schedule them. For this project, *the FreeRTOS Operating System* was used.
Specifically, both network-related tasks are pinned to *Core 0*. This because - using *FreeRTOS* on *ESP32* - the *event Task* is automatically pinned to *Core 0*, and so all WiFi events are dispatched on that core. On the other hand, the *LED Task* and the *Temperature Measuring Task* are left unpinned, delegating to the OS the responsibility to balance load and maximize throughput.
For what concerns communication, since the system relies on the *MQTT* protocol, the *PubSubClient library* was used. This particular library allows the programmer to subscribe to specific topics from a specific broker, without having to deal with protocol-specific low level aspects. The message content is represented in JSON format like it was for the *Window Controller*. In particular, the back-end sends periodically sends this sub-system the requested sample frequency, and in response this component sends back the current temperature measure, together with the date and the time of the measurement, for history-keeping reasons.
To synchronize the ESP time with current date and time I relied on the *United States National Institute of Standards and Technology time server*, as suggested by *this guide page*.

3.2 Window Controller

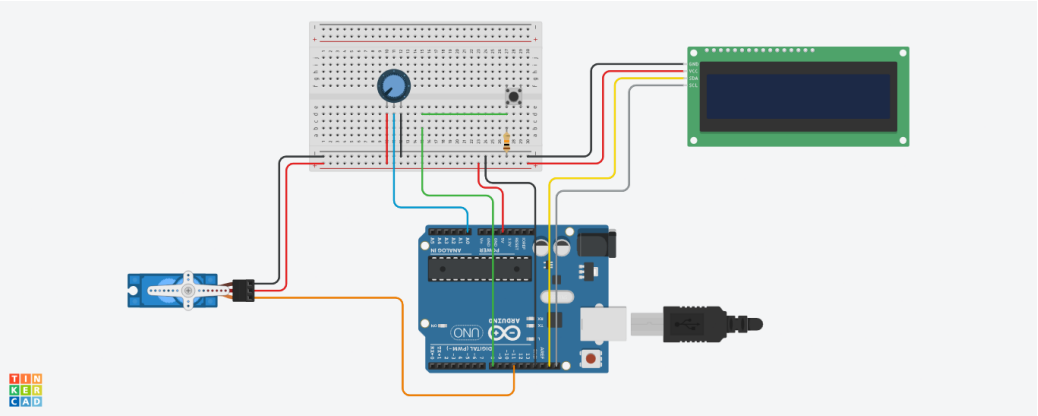

Figure 3.2: Detailed circuit for the window controller sub-system

In this scheme, available at *this link*, is depicted the detailed circuit for the *Window Controller*, deployed on an *Arduino UNO Board* and programmed relying on the *Wiring* framework.

One of the main concerns here is to implement our own scheduler, since Arduino doesn't support any kind of Operating System. This can be easily achieved implementing a cooperative round robin scheduler, since all Tasks have finite - and relatively short - execution routines. In addition to that, this kind of scheduler also defines implicit priorities among tasks, which can be used to the developer's advantage deploying tasks in such a way to avoid invalid data reads (e.g. collecting input and forwarding it to the *Control Unit* after having actuated the consistent data collected earlier). Attention must be paid to defining the Tasks' periods, in order to avoid exceeding deadlines on one hand and event loss on the other.
Another concern is the data format to send on the *Serial Line*. All messages are formatted in JSON notation, in order to be consistent with the other components' communication standard. From the *Control Unit* to the *Window Controller*, the message contains the last measured temperature value, the current control mode (automatic or manual) and the current opening level to be actuated. On the other hand, the *Window Controller* sends a JSON object containing two values, representing respectively if a mode switch was requested, and the opening level requested by the in-place operator. The handling of those two values is delegated to the *Control Unit*, which is the one coordinating all sub-systems and storing the valid representation of the system's state at any time.

3.2.1 Libraries and External Dependencies

For the scheduler timing purposes, I relied on this *Timer library* to avoid relying on system timer interrupts.
The LCD Screen, which works with the I^2C protocol, is controlled by the *LiquidCrystal* I^2C *library* to avoid having to handle all the low-level aspects of that communication protocol.
The servo motor is controlled via the *Servo Timer 2 library*, which exploits one of the 8-bit system timers leaving the only 16-bit timer (Timer 1) free for other purposes.

3.3 Operator Dashboard

This sub-system is a web-based application that allows an operator to inspect the current state of the system, and to perform some simple actions on it. The core of this system is the user interface, realized with *HTML, CSS* and *JavaScript*.
In particular, *HTML* and *CSS* are used to structure the page, and are re-

trieved whenever a new dashboard window is open, and then *JavaScript* handles automatic refresh and user operations. Automatic refresh is needed to retrieve - via *HTTP GET* requests at a specific *URL* - the data to be shown to the operator, while on the other hand user input is handled via *HTTP POST* requests to send data to the *Control Unit*.
Like other sub-systems, it considers only the system state sent by the *Control Unit* to be consistent, but it also keeps a cache of the history of the system in order to properly show the history graph to the user: it will be empty on startup, and it will be gradually drawn while refreshing the page increasingly adding the new data it receives.
To draw the measurements history, the *Chart.js library* was used.

3.4 Control Unit

The *Control Unit* is the core sub-system, that keeps track of the whole system's state in a consistent way. It's on its own divided in a few parts.

3.4.1 Database

The first component is a rather trivial, yet essential, database implemented in *mySQL*. It's made out of only one table, but it allows the system to save data persistently and to keep an history of the measurements even if the *Control Unit* stops its execution.

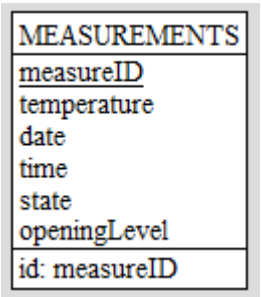

Figure 3.3: Scheme of the database table

3.4.2 Serial Agent

This component actively communicates with the *Window Controller*. It's implemented in Java, relying on the *JSSC library* for Serial Line Communication. It's responsible of gathering the in-place control unit input, com-

municating it to the *Central Controller* in order to reply effectively to the *Window Controller* to actuate the - eventually - mutated state of the system.

3.4.3 MQTT Agent

This component actively communicates with the *Temperature Monitor*. It's also implemented in Java, relying on the *VertX library* to implement an event-based asynchronous component. Its main activity is to wait for the *Temperature Monitor*'s messages, replying with the - eventually - mutated state of the system after communicating the measurements to the *Central Controller* and retrieving the new state of the system.

3.4.4 HTTP Server

This component acts as a very simple Server to answer the *HTTP* requests coming from the *Operator Dashboard*. It's also implemented in Java relying on the *VertX library*. Similarly to the *MQTT Agent*, it waits for the *Operator Dashboard*'s requests, retrieving then valid data from the database thanks to the *Central Controller*, and then communicating it as a response.

3.4.5 Central Controller

This component is the actual core of the whole system. It's responsible of all access to the database, and therefore must implement strong thread-safe procedures in order to ensure that data is handled properly. It's implemented in Java, relying on the *native JDBC API* to connect to the database and query it for data management.