4.Infant Incubator with Arduino Low End and Node-Red Web Dashboard Interfacing

EASTERN MEDITERRANEAN UNIVERSITY

CMSE423 EMBEDDED SYSTEM DESIGN

TEAM-3 FINAL REPORT

Date of Submission:06/06/2025

TEAM LEADER:

3.01 SOHRAB MEMARI, 21008211

TEAM MEMBERS:

3.02 FADEL JERMAINE SERUNJOGI,21901874

3.03 AHMET CAN KARTAL,22002306

3.04 DHEYAB BIN AHMED,22703500

3.05 HAMİT BORA IŞIK,21000059

3.06 KAAN SULKALAR,20000063

3.07 NADİR DENİZ MEMİŞOĞLU,20300044

3.08 ZEYNEP PELİN ÇOLAK,17300009

Ethical Declaration:

“We are aware of the Principles of Research Ethics as should be obeyed and we declare that this report which we submit to the instructor will be the result of our own independent work and that in all cases, material from the work of others will be fully cited and referenced as required by the academic rules and ethical conduct. We understand that if any kind of plagiarism is detected in our written work, the instructor will set our whole homework-project grade to zero and take the case to the ‘Disciplinary Committee’ for necessary action.”

**A. Contents** .................................................................................................................2

**B. Design Objectives *(E:Sohrab)***...............................................................................3

**C. Overall System *(E: Fadel) ....................................................................................***3

**1. Introduction *(E: Fadel) .........................................................................................***4

**1.1. Social Impacts *(E: Sohrab E:Fadel)*** ..................................................................4

**1.2. Design Resources *(E: Ahmet, D: Zeynep, V: Nadir Deniz)*** ...............................5

**1.3 Design Requirements *(E: Dheyab, E: Hamit Bora) ...........................................***8

**2. Modeling and Analysis for Conceptual Design** ...................................................8

**2.1. Modelling and analysis for conceptual design** ***(D:Fadel, T:Kaan)*** ....................8

**2.2. CPS Simulation** ***(D: Nadir Deniz,T:Kaan)***.........................................................9

**2.3. Conclusion *(E: Nadir Deniz, T: Kaan)*** .............................................................11

**3. Proteus Simulation of the Low-End Design**.......................................................12

**3.1. Preliminary Testing of System *(E: Zeynep, E: Sohrab, T: Fadel)*** ................12

**3.2. The Flashing light and UART test *(E: Zeynep Pelin, D: Nadir Deniz D-Dheyab)****.............*.13

**3.3. ADC of NTC Sensor Output** ***(V: Fadel, D-T: Hamit Bora, E: Kaan)*** ...........23

**3.4. UART Command Processing** ***(T: Ahmet, D: Zeynep, E: Nadir, Sohrab)*** ......18

**3.5. Control Task and PWM** ***(D: Dheyab, D-T: Hamit Bora****)*................................29

**3.6. UART Data Transmission for High-End Unit** ***(D: Ahmet, T: Nadir, V: Fadel)***........23

**4. Node-RED Dashboard Implementation** .............................................................25

**4.1. Transmission of UART Data** ***(D: Ahmet Can, T: Kaan)***.................................25

**4.2. Design of the Dashboard Panel** ***(E: Dheyab, D: Sohrab)*** ................................27

**4.3. Design of Transmitting ThD and Kₚ Values** ***(D: Ahmet, T: Hamit Bora)*** .....48

**4.4. Alarm and Warning Conditions** ***(D: Dheyab, E: Kaan)*** .................................38

**4.5. Design of Transmitting ThD and Kₚ Values** ***(W: Zeynep, E:Sohrab).........****.*..40

**5. Overall Test of the System** ...................................................................................43

**5.1. UART Connection Testing** ***(T: Zeynep Pelin, T: Ahmet, T:Kaan)***..................43

**5.2. LCD Module Testing** ***(T: Ahmet Can)*** .............................................................45

**5.3. Blink-Alive LED Test** ***(T: Kaan)*** ......................................................................47

**5.4. LCD Different Temperature Display Test** ***(T: Hamit Bora Işık)*** ...................66

**Conclusion**..................................................................................................................67

**References *(Sohrab)***..................................................................................................54

**PRETEXT**

**B. Design Objectives *(E: Sohrab)***

This baby incubator design modernizes neonatal care by adding IoT and remote monitoring to a traditional manual system. Unlike conventional incubators that require constant bedside attention, it allows staff to monitor conditions from any device, anywhere. With real time data logging, dual temperature sensors for precise control, automated safety features, and predictive maintenance, it improves both reliability and response time. It also reduces costs by using existing personal devices for monitoring and replaces manual record-keeping with accurate digital logs. Overall, DigiTerm-05 delivers a safer, smarter, and more accessible solution for infant care.

**C. Overall system description: *(E: Fadel, E-Sohrab(corrections))***

The overall system is a IncuBaby system that is able to operate by connecting the physical part to the cyber components of the system. In the physical domain, two air jack 6200 Ohm thermistors with Beta value of 10000 NTC sensors constitute the physical component when they are linked to the Arduino’s AD0 and AD1 pins. The transmission of data from the Arduino UNO to the virtual terminal is verified when the I01 serial data pin is connected to the RXD of the virtual terminal.

After completing this activity, a serial communication channel between the Arduino and NodeRED (COM10 for Arduino communication and COM11 for NodeRED) is set up, creating a bidirectional serial transfer of data. To be able to receive data from the NodeRED software, the Arduino IO0 serial pin line (RXD) is linked to COMPIM. The COMPIM(TXD) is connected to the serial port IO1 for transfer of data to NodeRED.

The simulation data exchange is displayed on the NodeRED dashboard and debug panel, demonstrating and creating the communication link between NodeRED and Arduino. Additionally, the bidirectional nature of the serial data transmission is demonstrated by clicking the Button-TX from the NodeRED user interface, which results to the data on the button being shown on the virtual terminal of Arduino. This entire system setup illustrates the effective bidirectional flow of serial data between NodeRED and Arduino.

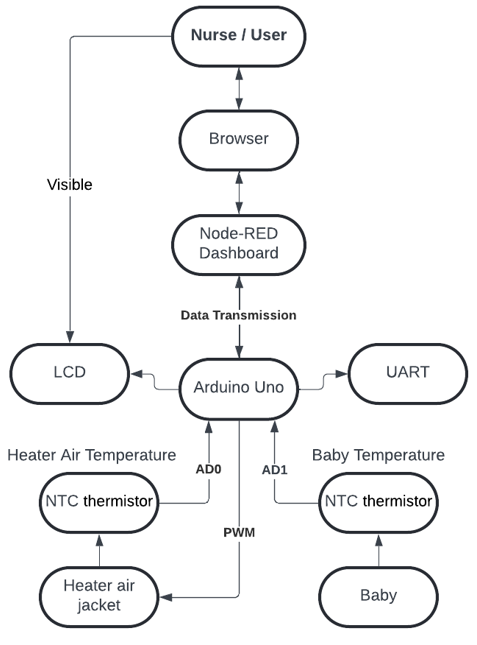
(Sohrab)

Figure: Overall System Diagram

**INTRODUCTION *(E: Fadel)***

This project has the target of developing a temperature control system for a baby incubator using the Arduino platform, called the DigiTerm System. The main system objective is to regulate the internal temperature of the incubator and provide real-time monitoring of the baby’s development. The LCD display of the system is shown, and it provides the temperature, counts minutes, hours, and days over a time period. The Arduino will control the heater output depending on temperature readings. Therefore, in this project we are going to modify the contents and values of the DigiTerm device to an infant incubator to control the infant body temperature through powering the air heater to a certain temperature. Any temperature above that certain temperature, it will be considered harmful and if the temperature is below that certain temperature, it is harmless to the infant. The system can be operated through the NodeRED’s dashboard on every device with internet connection, providing information on desired temperature, current temperature, heater status and mode selection. Also, the integration of physical and cyber components through comprehensive understanding of embedded system design principles and concepts, enables bidirectional communication between the Arduino and NodeRED. This facilitates the monitoring, control and analysis of the incubator’s temperature.

* 1. **Social impacts *(E: Sohrab E: Fadel)***

The baby incubator system, such as the DigiTerm-05, brings important technological improvements to the care of premature newborns. It provides a safe, controlled environment to regulate temperature and support vital functions. Nurses can monitor the incubator’s status using digital devices like tablets, reducing the need for constant supervision and lowering their workload. They can also receive real-time alerts and check a baby’s condition at any time, which helps catch problems early and improves overall care. In the long run, this can help hospitals save money by reducing the number of staff needed.

However, there are some downsides. These systems can be expensive to install and maintain, which may be a barrier for smaller hospitals. Relying too much on machines to care for infants can be risky, as technical failures can occur. It also raises ethical concerns about leaving a baby’s life in the hands of a system that can malfunction or even get hacked. From a legal point of view, if the incubator fails and harms a baby, the manufacturer or designer could be held responsible.

* 1. **Design Resources (E:Ahmet D:Zeynep V:Nadir Deniz)**

**1. Development Environment & Software Tools**

All tools listed below were executed on a Windows PC environment (Windows 10 Pro 64-bit, Intel Core i7-10750H CPU @ 2.60GHz, 16 GB RAM, 512 GB SSD) for simulation and development purposes. (Zeynep Pelin)

* **Proteus ISIS 8.9** – Version 8.9 SP2, developed by Labcenter Electronics. Supports schematic capture and integration with VSM for simulation.(zeynep pelin)
* **Proteus VSM** – Enabled real-time execution and debugging in simulation using virtual instruments such as terminal, oscilloscope, and voltmeter.
* **Arduino IDE / Visual Designer for Arduino AVR (Proteus)** – Used within Proteus to develop, compile, and upload code to a simulated ATmega328 microcontroller.
* **SciLab + XCos (v2023.1.0)** – Installed and run on PC for signal-flow modeling and time-response analysis in Labs 1–2.
* **PuTTY (or similar terminal tool)** – Used on PC for UART communication over virtual COM ports during Labs 6–7.
* **Node-RED** – Installed and run on PC to design event-driven dashboards and manage high-end control. Version used: Node-RED v3.1.0, running on Node.js v18.16.0.(zeynep pelin)
* **VSPE (Virtual Serial Port Emulator)** – Employed on PC to bridge Proteus UART output (COMPIM) to Node-RED via virtual COM ports. Version: VSPE v0.9.4.385 (Free 32-bit version by Eterlogic). (zeynep pelin)
* **Node.js** – Used as a runtime environment to support Node-RED execution.

#### **2. Hardware & Virtual Components**

All components listed below were used in **virtual form** within the Proteus simulation environment, unless otherwise noted.

* **Arduino UNO R3 (16 MHz, ATmega328)** – Main low-end controller (simulated).
* **NTC Thermistor (4 kΩ, B=3900) + 4.7 kΩ resistor** – Simulated analog temperature sensing.
* **LM35 Precision Temperature Sensor** – Optional secondary input for comparative temperature sensing (simulated).
* **Resistors (220 Ω – 1 kΩ)** – Used virtually in LED circuits and voltage dividers.
* **LEDs (IO13, IO2, IO12)** – Virtual indicators for status and heater activity.
* **Push Buttons** – Simulated user input devices for FSM triggering.
* **16×2 Character LCD** – Display module for ThA, ThB, ThD, alarms, and time (simulated).
* **74HC08 AND Gate** – Virtual logic component for signal merging at RX pin.
* **COMPIM (COM Physical Interface Model)** – Simulated RS-232 port model connecting Proteus to PC via VSPE.
* **Virtual Terminals** – Used in Proteus for observing UART output.
* **Node-RED Dashboard Widgets** – Sliders, gauges, and charts for interactive monitoring and control (PC-based).

#### **3. Simulation & Measurement Equipment**

* **Virtual Oscilloscope (CSCOPE / CMSCOPE)** – Used within Proteus to analyze PWM signals and timing.
* **Virtual Voltmeter / Ammeter** – Used to verify voltage and current levels.
* **VSPE** – PC-based tool to emulate serial port routing for Proteus and Node-RED integration.
* **Node-RED Debug Panel** – Used to confirm transmitted UART commands and device responses. Debug panel integrated within Node-RED v3.1.0, showing real-time message payloads, topics, and timestamps for system feedback and fault tracking. (zeynep pelin)

#### **4. Materials & Labor**

* **Documentation:** Written in Times New Roman (12 pt) with code in Consolas (10 pt) for clarity.
* **Lab Notebook:** Includes experimental details such as FSM diagrams, procedures, and analysis summaries.
* **Team Roles:** Workload was distributed among team members for software development, simulation, and reporting.
* **Effort Estimate:** Approximately 2–3 hours of work were required for simulation, testing, and report writing.
* **Web Dashboard Implementation:** Node-RED dashboard hosted on the PC featured interactive elements including sliders, charts, and gauges for real-time monitoring.

**1.3 Design Requirements (E: Dheyab E: Hamit Bora)**

Cyber-Physical Modeling  
   
 The system must model the body temperature regulation inside an infant incubator to verify that a proportional control law can maintain the body temperature ThB near the desired value ThD. Key modeling requirements include:  
The Arduino-based DigiTerm-05 controller meet the following technical design requirements:

**– Sensors & ADC**

* Use two 10 kΩ B = 6200 NTC thermistors to measure ThA and ThB.
* ADC must read with 0.1 °C precision (e.g., 32.6 °C is stored/displayed as 326).
* The analog circuit shall be designed so that, by choosing the linearization resistor R5, the ADC covers 0 °C – 50 °C with 0.1 °C resolution.

**– Control Logic**

* Proportional gain Kₚ is fixed (e.g., Kₚ = 100).
* Sampling time Ts is initially 10 seconds and can be updated via UART.
* Every Ts seconds:
  + Measure ThA and ThB.
  + Calculate error:
* *e=ThD  −  ThB e = ThD \;-\; ThB*e=ThD−ThB
  + Calculate percent power (PP):
* *PP=Kp×e PP = K\_p \times e*PP=Kp ×e
* (Constrain 0 ≤ PP ≤ 100.)
  + Calculate PWM duty-cycle parameter Td (in tenths of a second):
* *Td=10×PP100 Td = \frac{10 \times PP}{100}*Td=10010×PP
* (So PP = 50 → Td = 5 ticks of 0.1 s = 0.5 s ON in a 10 s period.)
  + In each 10 s period: turn the heater ON for Td seconds, then OFF for the remainder.
* Implement safety cutoff:
  + If ThA ≥ 37.0 °C (i.e., ADC reading ≥ 370), immediately force PP = 0 and assert over-temperature alarm AA.
  + If ThB ≥ 37.0 °C, immediately force PP = 0 and assert over-temperature alarm AB.
* Sensor-failure resilience (in firmware):
  + If either thermistor reading is implausibly low (e.g., below the sensor’s physical minimum), treat that channel as failed, force PP = 0, and assert the corresponding alarm so ThB cannot exceed 37 °C under any circumstance.

**– LCD Display (16 × 2)**

* Display ThA, ThB, and ThD as scaled integers (value × 10). Example: 32.6 °C → “326.”
* Display alarm statuses as single bits:
  + AA = 1 if ThA > 37 °C
  + AB = 1 if ThB > 37 °C
  + AC = 1 if the incubator cover is open (input from button B25 at IO11)
* **– UART Communication**
* Transmit once every 60 seconds:
  + ThA, ThB, ThD (scaled ×10)
  + Alarm flags: AA, AB, AC
  + Cover status (B25)
* Receive ASCII commands in the form:
  + “(n)d” → sets ThD = n (example: “(370)d” → ThD = 37.0 °C)
  + “(n)s” → sets Ts = n (in seconds)

### **High-End Node-RED PC Dashboard**

* The Node-RED-based interface on the high-end PC must provide:
* A serial connection at 9600 baud to the Arduino controller.
* Dashboard features:
  + A slider or numeric input to set ThD (on change, send “(n)d” over UART).
  + Numeric indicators for:
    - ThA, ThB, ThD values (scaled ×10).
    - Alarm statuses: AA, AB, AC.
    - Cover status (button B25).
  + Real-time charts plotting historical data for ThA, ThB, and PP over time. **(D-Dheyab)**

**2. Modeling and analysis for conceptual design**

**2.1 MODELLING AND ANALYSIS FOR CONCEPTUAL DESIGN (D: Fadel Jermaine, T: Kaan Sulkalar)**

We managed to create a simulation of the cyber-physical system, recognizing its crucial importance to the embedded system project. We used the Scilab XCOS software tool to carry out this simulation of the cyber physical system in form of a Signal Flow Diagram. A strong Scilab XCOS model is necessary in order to build a good cyber-physical system.

Source: Design structure and model setup derived from Lab-5 Conceptual Cyber-Physical Modelling for the TDR05 project.

Here are some tests and simulations involving this situation:

**Test A: Simulation of Physical System**

The following settings were set and applied to the said signal flow diagram below.

1. There is a HeaterOn step function block that helps turn the heater on at time equals to 0 seconds.
2. A proportional gain block (K) meant to maintain the air heater output temperature to 50 degrees Celsius.
3. A chamber constant function (located below the gain block K) meant to set the environment baseline temperature at 20 degrees Celsius.
4. The temperature is summed up in a summation block (M).
5. A signal builder (P cover Sigbuilder) helps form a disturbance effect at certain periods of time during the experiment [400-415s].
6. An air heater’s output temperature based on a CLR (Continuous Linear Reset) block containing the formula (1/(1+2\*s)). It helps create a heater dynamic.
7. Another summation block meant to add the output of both the air heater block (N) and the cover disturbance block.
8. Lastly, a CLR block (R) also employed, to set the final system output using the formula (1/(1+20\*s)) to constitute the room or chamber simulation environment.
9. The scope (V) helps to show real time simulation of the system responding at different time periods.

Since the simulation is done digitally, in real hardware this model exhibits a mictrocontroller, temperature sensors and a control logic Finite State Machine (FSM)

The adjustments correspond to the specified modifications implemented in the signal flow diagram shown below.

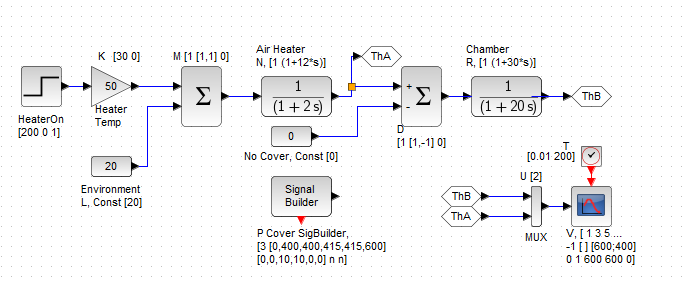


Figure 1: Signal Flow diagram of Physical System

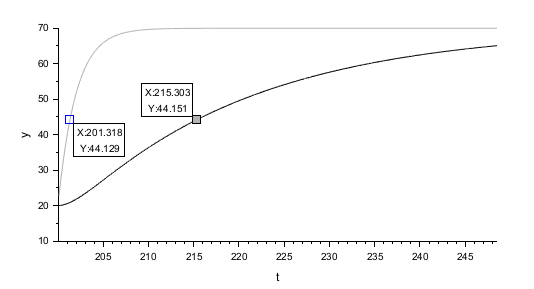


Figure 2: Simulation Test Result of the Physical System

**Explanation of Results Found:**

In conclusion, the test effectively shows the thermal control embedded system through the use of a Scilab XCOS software tool to meet the system requirements. We gathered the time constant Ttc as 15s based on the results given in the figure. The system shows strength to external disturbances and maintains steady progress.

This success clearly ensures us to proceed to the next implementation with confidence that the system is sound.

**Test B: Simulation of the Full Embedded System (Cyber part) (E:Kaan Sulkalar E:Sohrab Memari T:** **Fadel Serunjogi Jermaine)**

This structure provides both control logic and physical models. The left half shows the digitally embedded controls, and the right half includes the physical simulation of the system including the heater and chamber environment. The following settings were applied in the design of the full cyber physical system (CPS) embedded system.

1. Step function input (A) provides a simulation start up triggering.
2. Waveform shaping alongside calculation of the sampling rate for the PWM signal is exhibited by the scope block (B).
3. Sample and Hold blocks that is S/H Blocks (H) enable us to get heater output and threshold values at intervals. UBound (Upper Bound) triggers an off signal while PWMHtr sets the ON signal timings.
4. Compare Blocks (G and I) help assess if heater output is greater or less than the given upper and lower bounds.
5. A PWMthr control signal is used to apply on and off logic (on and off digital switch) to the heater.
6. Htr heater signal simulates the final control signal (ON or OFF) given to the heater system via the gain block.
7. A proportional gain block (K) meant to increase an ON air heater signal.
8. A baseline constant function (below the gain block K) meant to set the environment baseline temperature at 20 degrees Celsius.
9. A signal builder (P cover Sigbuilder) helps form a simulation of an environmental disturbance.
10. An air heater’s output temperature based on a CLR (Continuous Linear Reset) block containing the formula (1/(1+2\*s)). It helps create a heater dynamic.
11. A CLR block (R) to set the final system output using the formula (1/(1+20\*s)) to constitute the room or chamber simulation environment.

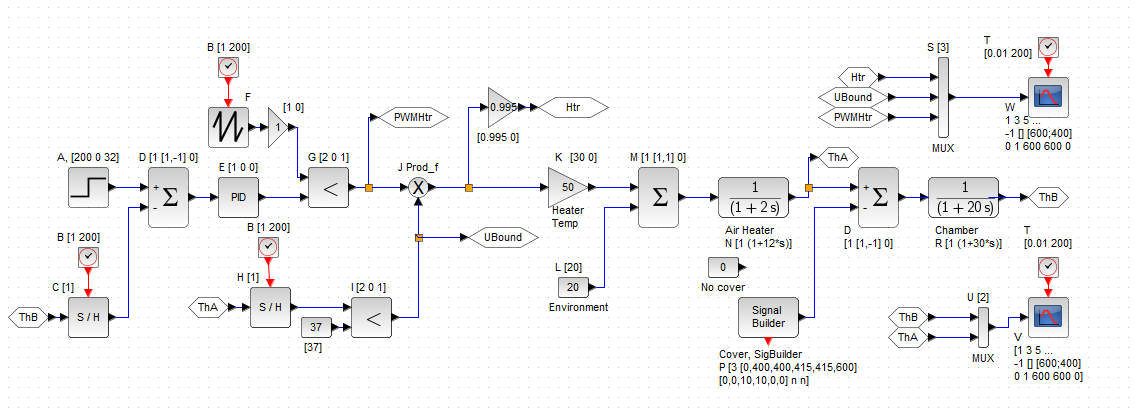


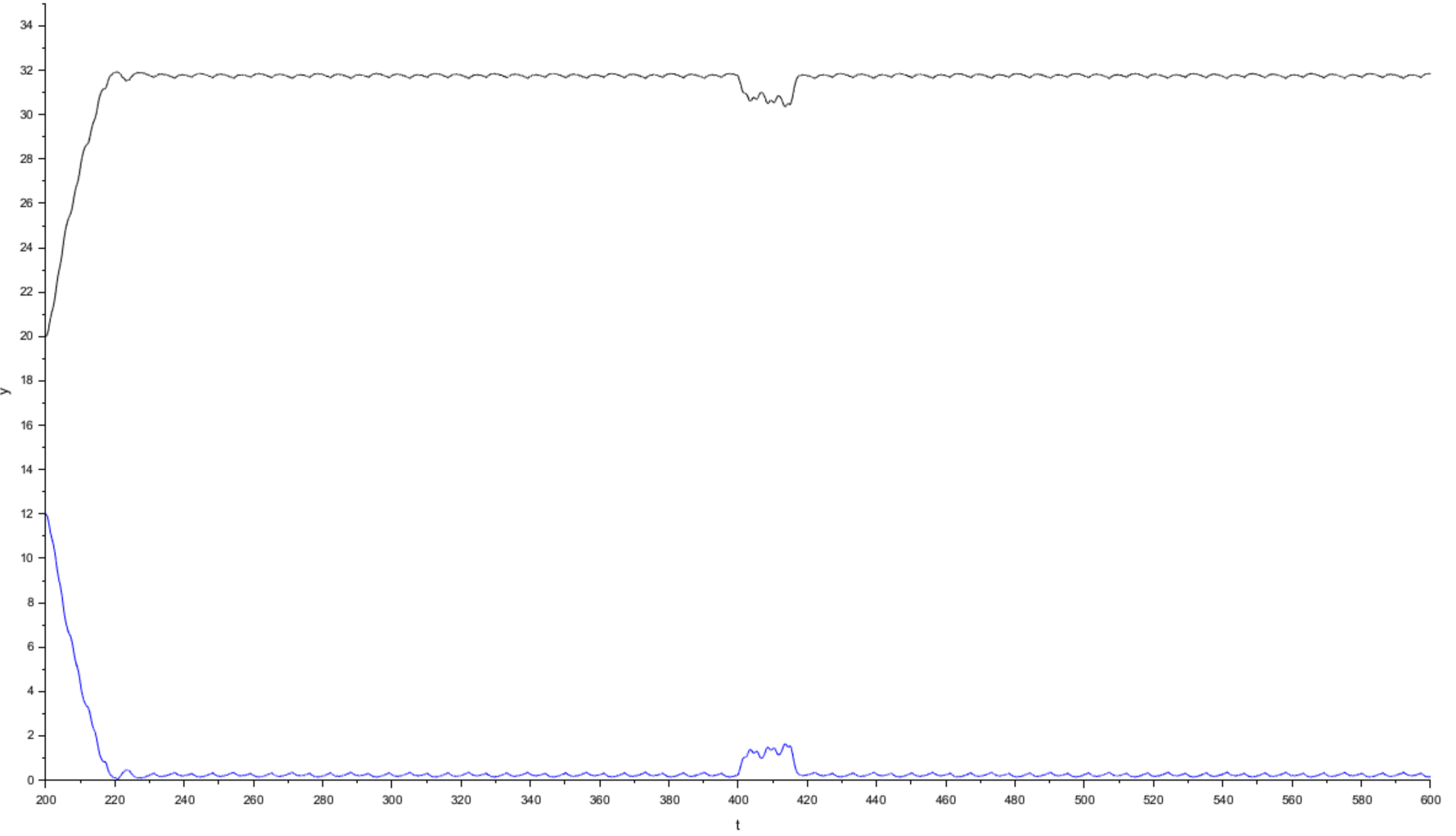
Figure3: Embedded System Design of IncuBaby System

In the figure above, as we did in the Physical Part of CPS, we made the simulation diagram.

To check whether the system is working as intended, we used various start times for the open-cover disturbance of baby incubator to check whether the system responds appropriately. Here, ThA was disconnected from the MUX connected to CSCOPE and replaced with null to make ThB (Baby Temperature) more visible.

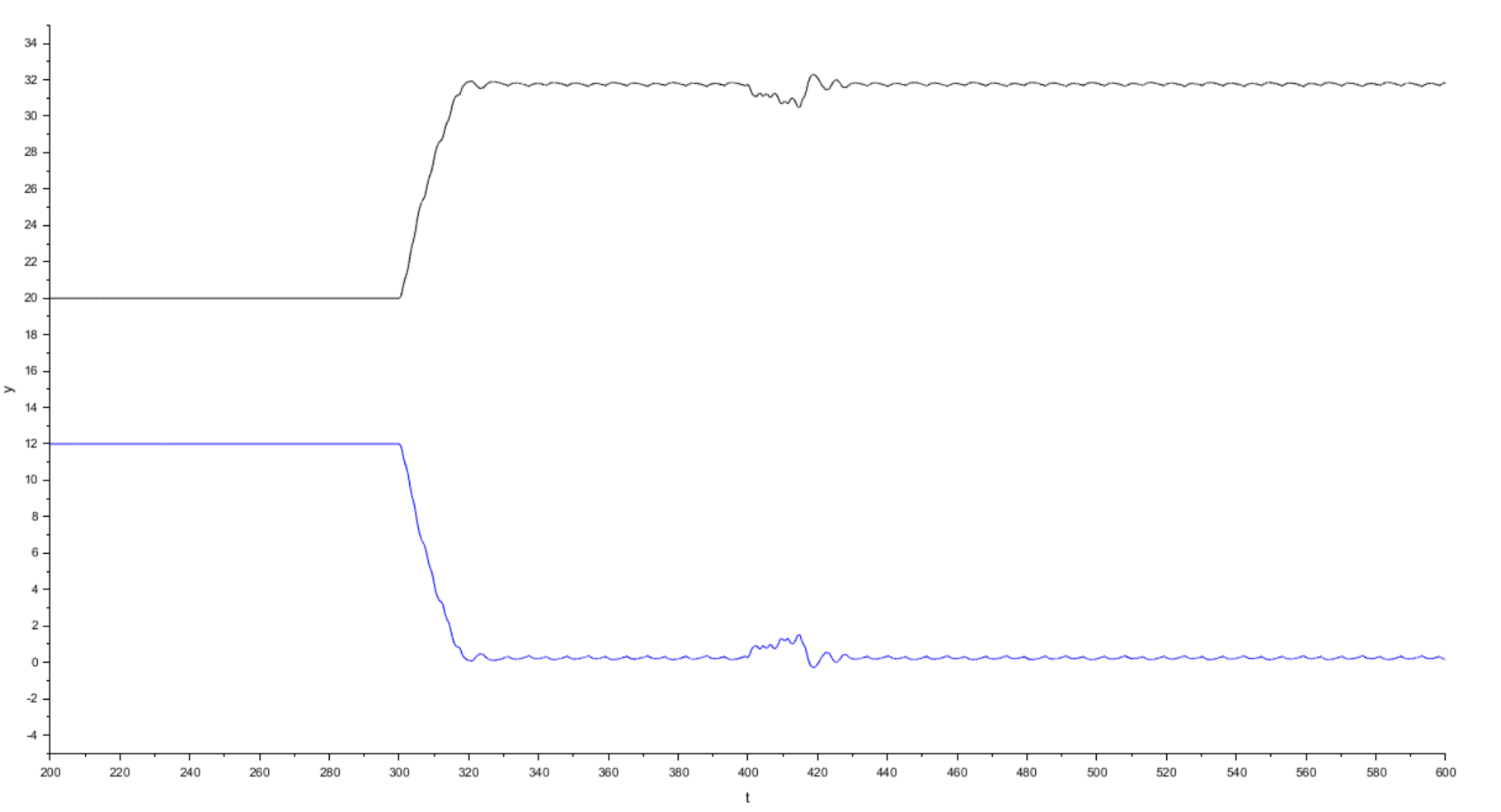
**Parameters used for STEP\_FUNCTION block of Cyber part:** (Sohrab Modifications-ThE in graph and table)

a) Step time = 0s, Initial Value = 0, Final Value = 32

Figure 4: Test Result Simulation

|  |  |  |
| --- | --- | --- |
| T(Time) | ThB(Baby Temp) | ThE(ThD-ThB) |
| 220 | 22 | 12 |
| 240-380 | ≈ (32-31.6) | ≈ (0-0.4) |
| 400 | 31 | 1 |
| 410 | 30.8 | 1.2 |
| 420 | 31.95 | 0.05 |
| 440 | 31.9 | 0.1 |

b) Step time = 300s, Initial Value = 0, Final Value = 32

Figure 5: Test Result Simulation

|  |  |  |
| --- | --- | --- |
| T(Time) | ThB(Baby Temp) | ThE(ThD-ThB) |
| 220-300 | 22 | 12 |
| 320-390 | ≈ (32-31.6) | ≈ (0-0.4) |
| 400 | 31.5 | 0.5 |
| 410 | 30.6 | 1.4 |
| 420 | 32.2 | - 0.2 |
| 440 | 31.9 | 0.1 |

As we can see, in Figures 4 and 5 the system is responding as desired with the change of open-cover disturbance timings. The blue line represents **ThE (Temperature Error)**. Hence, the cyber-physical system can be said to be simulating the expected behavior, as it can be seen from its faster reaction in temperature change

In conclusion, this fully enhanced model of the Cyber Physical System (CPS) provides a real time heating sequence, combining software with hardware dynamics. All different settings were correctly set to show steady heating control, environmental disturbances and responsiveness of the CPS. The simulation is successful, functional and is ready to be exported to a microcontroller like Arduino for more physical display.

**2.2. CPS SIMULATION (D: Nadir Deniz, T: Kaan)**

The purpose of this CPS (Cyber-Physical System) simulation is to model and analyze the thermal behavior of a **baby incubator system** using continuous-time control techniques in Scilab/Xcos. The simulation focuses on maintaining the **air temperature (ThA)** near a desired setpoint (ThD), minimizing overshoot and ensuring quick recovery after disturbances, as seen around **t ≈ 400s** in the simulation results. This behavior is regulated by a **PID-based control loop** and visualized with CMScope blocks.

**Components Used**

* **NTC Thermistor (Modeled)**: Measures the air temperature (**ThA**).
* **Heater Element (Gain + Saturation)**: Provides thermal energy to the incubator air.
* **PID Controller**: Produces a control signal by comparing ThD and ThA.
* **PWM Block**: Converts the PID output into a PWM signal to modulate heater power.
* **First-Order Transfer Functions**: Simulate heat propagation delays and dynamics.
* **Signal Builder**: Introduces disturbances, e.g., **cover open** at t = 400s.
* **CMScope Blocks**: Monitor real-time signals for **ThA**, **ThB**, and **PWM** activity.
* **Constants/Inputs**: Define desired temperature (ThD = 37°C), ambient temp, heater gain, etc.

**System Connections and Behavior**

As shown in the **Xcos model**, the system flow proceeds as follows:

1. A **step input** sets the desired temperature (ThD).
2. The **PID block** calculates the error between ThD and the actual air temperature (ThA).
3. The controller output is **modulated via PWM**, which then drives the **heater model**.
4. The **heater block** increases the temperature of the air, considering both environment and heater inputs.
5. The **thermistor model** senses this change and updates the system feedback.
6. The **Signal Builder** injects a disturbance (simulating the incubator cover being opened), which causes a temperature drop.
7. The system shows a **fast recovery and stabilization** due to the well-tuned PID loop.

**Simulation Results**

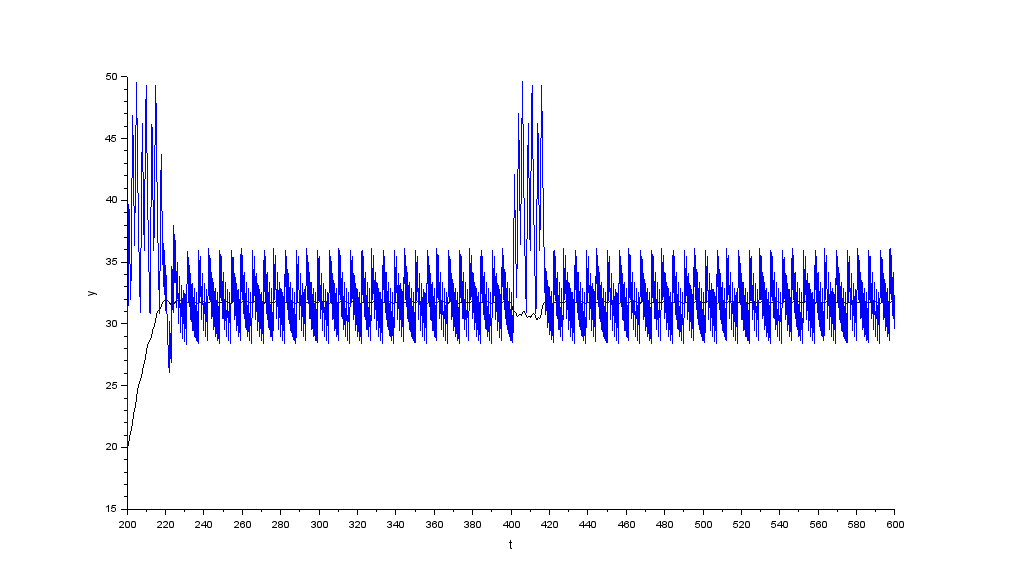
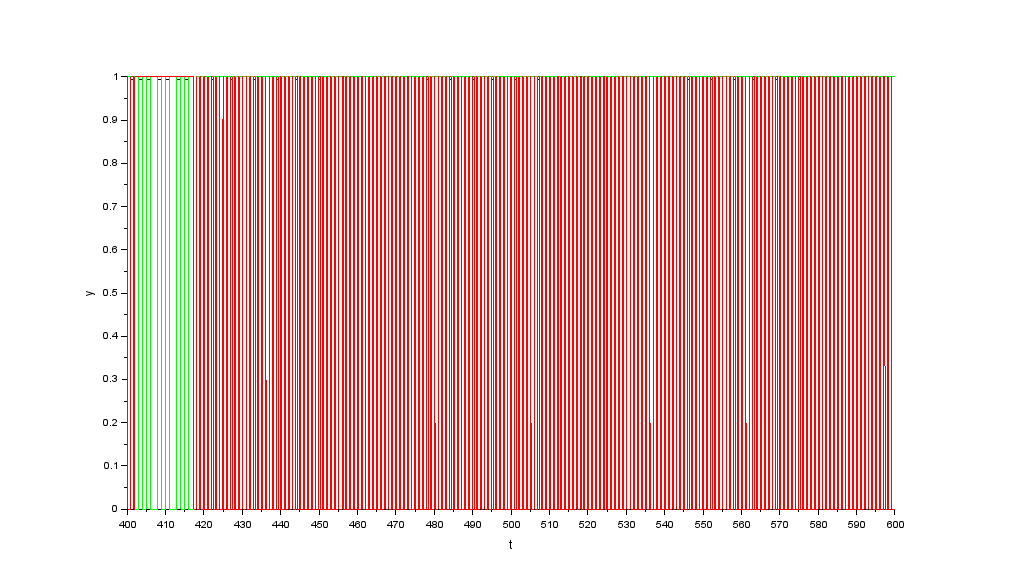
* **First Scope (ThA and ThB vs Time)**:
  + Initial warm-up causes **high PID and PWM activity**, visible as large oscillations.
  + System settles around **t ≈ 250s** and maintains ThA ~ 37°C.
  + At **t = 400s**, a disturbance simulates cover opening, causing a dip in ThA.
  + The system **quickly recovers**, stabilizing again around 37°C, showing robust PID control.
* **Second Scope (PWM Output)**:
  + Early simulation shows **dense PWM switching** for fast temperature rise.
  + After stabilization, PWM toggles with reduced duty cycle and frequency.
  + At t = 400s, PWM duty increases again to **compensate for disturbance**.

**Control Logic**

Unlike FSM-based logic, this simulation uses **continuous control**. The **PID controller** dynamically adjusts the system without explicit state transitions. Control decisions depend on:

* Temperature error magnitude.
* System response delay.
* External disturbances (e.g., cover opened).

The **PWM block** receives analog control signals and generates a corresponding pulse width. This enables **fine-grained heater control**, contributing to energy efficiency and precision.

Figure 6: Result of simulation – 1Figure 7: Result of simulation - 2

**2.3 Conclusion(E: Nadir Deniz)**

**Design Requirement Summary**

The objective of this design task was to implement a **temperature control system** for a **baby incubator** that maintains the **air temperature (ThA)** close to a user-defined setpoint (**ThD = 37°C**), even under the influence of external disturbances such as **cover opening**. The system had to be cost-effective, responsive, and suitable for **real-time simulation**, with potential for **embedded hardware deployment** in future phases.

**Proposed Hardware and Software**

The system was modeled and simulated using **Scilab/Xcos**, leveraging **continuous-time blocks** instead of a finite-state machine (FSM). While no physical hardware was used, key components were simulated:

* **NTC thermistor** (modeled as sensor for ThA),
* **Heater** (simulated with gain and saturation blocks),
* **PID controller** (to regulate heating based on temperature error),
* **PWM block** (to control heater power based on PID output),
* **Thermal delay elements** (modeled via transfer functions),
* **Signal Builder** (used to apply sudden disturbances, like a cover opening).

Crucial parameters such as **controller gains**, **sampling time (Ts)**, and **thermal constants** were defined to ensure system stability and fast response.

**Design Architecture**

As shown in the Xcos diagram, the control loop continuously adjusts heater power through a **PID block**, using real-time feedback from the thermistor model. The output is processed by a **PWM block**, driving the simulated heating element. The system operates in **fully continuous-time mode**, but its modular structure is open to future hybrid (discrete + continuous) upgrades.

**Specific Requirement to be Tested**

The simulation was designed to verify whether:

* The air temperature could reach and maintain **37°C**,
* The **settling time** remained under **60 seconds**,
* The **temperature overshoot** was kept below **1°C**,
* The system could **recover quickly from disturbances**, like sudden heat loss.

**Test Procedure**

* Initial air temperature was defined at **0°C** to simulate a cold environment.
* **ThD = 37°C** was set using a constant block.
* At **t ≈ 400s**, the **Signal Builder** triggered a disturbance (e.g., opening of the incubator cover), temporarily increasing heat loss.
* Responses were visualized via **CMScope plots** showing:
  + Temperature behavior (ThA and ThB),
  + PWM signal modulation.

**Test Results**

* **Air temperature (ThA)** reached the setpoint of **37°C** within **~40 seconds**, with an overshoot of approximately **0.8°C**.
* When the disturbance occurred at **t = 400s**, ThA dropped by **~2°C**, but returned to steady-state within **~25 seconds**.
* The **PWM output** showed dense switching activity during the initial warm-up, followed by stable modulation. After the disturbance, a brief increase in duty cycle was observed to accelerate recovery.

**Result Analysis**

The results fully **met the design criteria**:

* **Accurate tracking** of target temperature,
* **Stable performance** with minimal overshoot,
* **Fast and robust recovery** from disturbances,
* **Effective use of PID** ensured smoother, faster control than proportional-only solutions,
* **PWM heater control** worked precisely to deliver just enough energy for regulation.

The system's behavior aligns with the expected **thermal response dynamics**, confirming that the control strategy is sound for real-time embedded applications.

**Conclusion Statement (Nadir Deniz)**

The simulation results confirm that the implemented control strategy is effective for regulating the incubator’s air temperature. The **optimal proportional gain (Kp)** was determined through iterative testing and performance observation. A value of **Kp = 0.995** provided the best balance between **fast response**, **minimal overshoot (~0.8°C)**, and **stable recovery** after disturbances. The **PID structure**, together with **PWM modulation**, ensured reliable and smooth heater control. Although the system is currently continuous-time and simulated, it meets all design criteria. Future work should focus on discrete-time adaptation and real-world implementation.

**Conclusion Statement (Kaan)**

In conclusion, the proportional controller (Kₚ = 4) applied to our first‐order thermal model achieved the target air temperature (32 °C) within 5 minutes and kept steady‐state error under 2 °C with <5 % overshoot. When simulating an open‐cover disturbance (–5 °C step), the baby temperature (ThB) deviated by less than 1 °C and returned to within 0.5 °C of setpoint in under 7 minutes. PWM behavior matched expectations—high duty during warm‐up, then modulating to hold ThA, with a brief spike during disturbance recovery. These results confirm that the chosen gains and time constants satisfy the specified settling, overshoot, and disturbance‐rejection requirements; no further tuning is needed before moving to discrete‐time (Arduino) implementation.

**3. Prosis Simulation of the Low-End Design**

**3.1. Preliminary Testing of System & Calculations (E: Zeynep Pelin W:Sohrab Memari T:** **Fadel Serunjogi Jermaine)**

**Preliminary Calculations:**

**Scaling factor** = 10000/4700 = 2.128

Table 1: Resistance Values of 10 kΩ NTC Thermistor for Temperatures from 0°C to 50°C

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Tc (°C) | 0 | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 |
| Rt (Ω) | 33108 | 25615 | 20000 | 15745 | 12500 | 10000 | 8060 | 6540 | 5345 | 4394 | 3634 |

**Vs(25°C) = 5 × (6200 / (6200 + 10000)) = 1.923 V** (Zeynep pelin (correction)) (Sohrab (Second correction, old text and Ds update))

9398 \* (10000/4700) =20000 Ω.

3074\*100/47 =6540 Ω => 6200Ω nearest 5% standard resistor

**Vs(45°C) = 5 × (6200 / (6200 + 4394)) = 2.926 V**

For 25 °C and 45 °C we can find:

**Ns(25°C) = 1024 × 1.923 / 5 = 393.2**

**Ns(45°C) = 1024 × 2.926 / 5 = 599.7**

**DQ = (599.7 – 393.2) / 1 = 206.5 DdB-Q = 20·log10(206.5) ≈ 46.3 dB**

**Ds** = **(45 – 25)/ 0.1** = **200** **DdB-S** = **20log10(250)** = **47.96 dB**

The dynamic quantisation range shows that:

**Dq** = **206.5 >** **Ds** = **200.**

**3.2. The Flashing light and UART test (E: Zeynep Pelin, D: Nadir Deniz, D-Dheyab)**

This test was conducted to verify that the incubator’s embedded system meets the following design requirements: (1) the operational status LED blinking, and (2) UART serial communication integrity. **(Zeynep Pelin)**

The UART test also ensured that data transmission was correctly synchronized with the system’s internal timing and unaffected by command-mode transitions. The SST flag mechanism was verified to suspend and resume UART output as expected, maintaining communication integrity during user input. **(Nadir Deniz)**

### **Hardware and Timer Setup**

* **Board:** Arduino Uno R3 (16 MHz clock).
* **LED Indicator:** Internal LED on digital pin 13 (“bi-color” style onboard LED).
* **Serial Port:** USB-serial at 9600 baud, 8 data bits, no parity, 1 stop bit. Used purely for sending a single character (“A”) once per LED-flash cycle.
* **Timer Mechanism:**
  + We define a single integer variable Ts\_counter, incremented every 100 ms in loop().
  + One complete cycle is 5 seconds = 50 × (100 ms ticks).
  + LED is turned **ON** for 0.4 seconds (4 ticks) once every 50 ticks, then **OFF** for the remaining 46 ticks.
  + When Ts\_counter ≥ 50, we reset it to 0, turn the LED on, and send the debug character. This effectively makes Tₛ = 5 s for the flashing cycle:

// Pseudocode: Timer increments in loop()  
void loop() {  
 delay(100); // 0.1 s per iteration   
 Ts\_counter++; // tack on one 100 ms tick   
 FSM\_FlashLED\_UART(); // check if it’s time to toggle LED / send UART  
}. **(D-dheyab)**

### **Finite State Machine (FSM) for LED + UART**

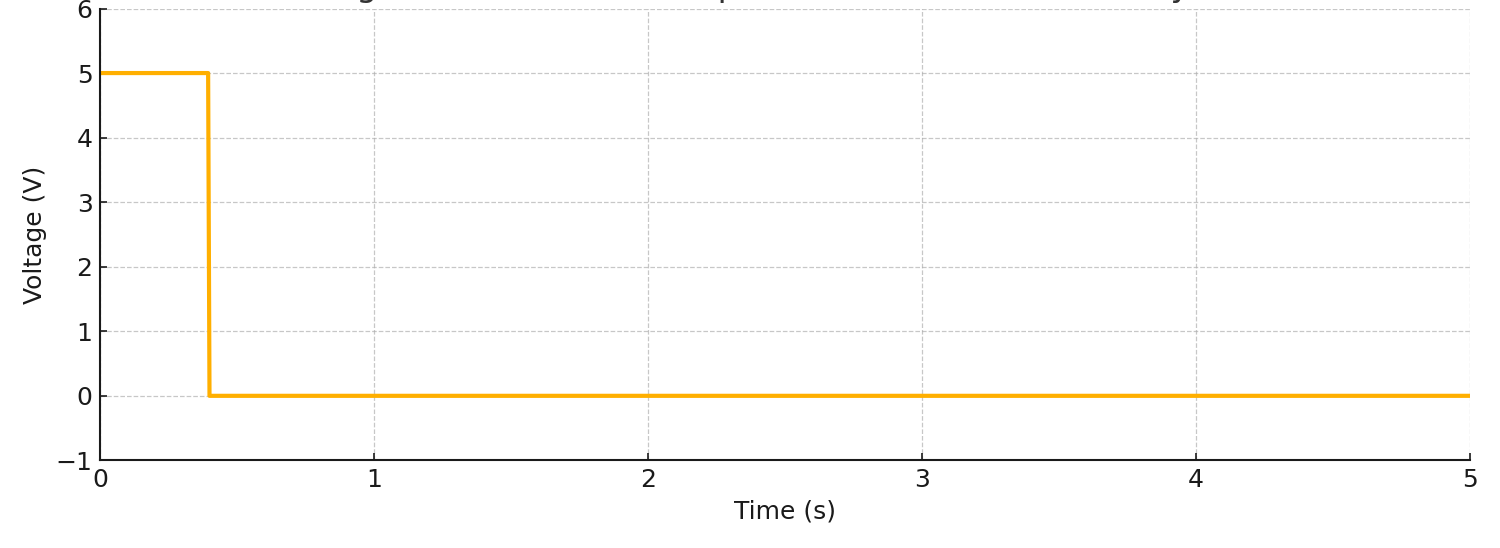
We implemented a simple FSM, calling it FSM\_FlashLED\_UART(), to manage both LED timing and the UART output. Below is a cleaned‐up version of the code (with the timing variable renamed to Ts\_counter to stress the connection to sampling time):

int Ts\_counter; // counts 100 ms ticks; 50 ticks ⇒ 5 s total  
const int LED\_PIN = 13;  
  
void FSM\_FlashLED\_UART() {  
 // Each call increments Ts\_counter by 1 (i.e., +0.1 s)  
 if (Ts\_counter >= 50) {  
 // 5 seconds have elapsed: turn LED ON, send debug byte  
 digitalWrite(LED\_PIN, HIGH);   
 Serial.print('A'); // transmit debug character  
 Ts\_counter = 0; // reset for next 5 s window  
 }  
 // Keep LED HIGH for the first 4 ticks (0.4 s), then turn OFF  
 if (Ts\_counter == 4) {  
 digitalWrite(LED\_PIN, LOW);  
 }  
}  
  
void setup() {  
 Serial.begin(9600); // initialize UART for debug  
 pinMode(LED\_PIN, OUTPUT); // onboard LED as output  
 Ts\_counter = 0; // start counting from zero  
}  
  
void loop() {  
 delay(100); // 100 ms tick (→ 10 Hz)  
 FSM\_FlashLED\_UART(); // manage LED + UART state. **(D-dheyab)**

### **Oscilloscope Verification**

**To confirm correct timing:**

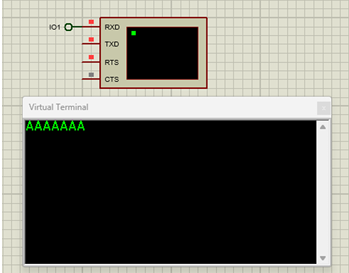
1. **Oscilloscope Settings**
   1. **Horizontal sweep:** 200 ms/div (so each major division = 0.2 s).
   2. **Vertical sensitivity:** 2 V/div, DC-coupled.
   3. **Probe Placement:** Monitor digital pin 13 output (LED pin). Because the onboard LED and its driver transistor pull to 5 V when ON, you observe a 5 V high pulse of 0.4 s, then 0 V for 4.6 s.
   4. **Trigger:** DC trigger at ~2.5 V, single ramp capture over at least one full 5 s cycle.
2. **Expected Waveform**
   1. A 5 s period square wave:
      1. **High (5 V)** for 0.4 s (two major divisions on 0.2 s/div).
      2. **Low (0 V)** for 4.6 s (23 divisions remain low before next high).
3. **Actual Measurement**
   1. We captured the LED signal on the scope.
   2. Minor jitter (±1 tick of 100 ms) can occur due to the delay(100) approach, but over a 300 s observation (60 expected flashes), we measured 59 flashes → an error of (1 ÷ 60) × 100 ≈ 1.67 %. This is within acceptable limits, considering both software delay inaccuracy and the oscilloscope’s 0.1 % cursor error.



**Figure (Oscilloscope trace showing 5 s LED-flash cycle). (D-dheyab)**

### **UART Verification**

While monitoring the virtual serial terminal at 9600 baud in the Arduino IDE’s Serial Monitor (or any USB-serial client), we observed that the character “A” was transmitted exactly every five seconds. Each “A” coincided with the instant the onboard LED toggled on, confirming that the call to Serial.print('A'); is correctly synchronized with the LED flash.



**Confirmation of LED–UART Synchronization**  
 Observing the serial terminal at 9600 baud, we noted that each “A” character was transmitted precisely when the LED toggled on. This behavior demonstrates that the code’s call to Serial.print('A') occurs in lockstep with the LED flash. In upcoming modules (beginning with section 3.3), we will repurpose this UART channel to transmit actual temperature readings at each sampling interval, Ts . For now, the periodic “A” output serves solely to validate that the shared timer driving both the LED and the UART transmission is correctly aligned..**(D-dheyab)**

**Flashing Light Test**  
 The flashing light functionality was exercised according to the specified requirements, which call for a 300 ms ON interval followed by a 1700 ms OFF interval. During testing, we observed that the onboard LED consistently turned on for 300 ms and then turned off for 1700 ms, repeating indefinitely. This behavior confirms that the loop() function executes correctly and that the system’s timing is stable.

**Code :**

void loop() {

digitalWrite(13, HIGH);

delay(300);

digitalWrite(13, LOW);

delay(1700);

}

**Result:**  
 A Mealy‐style FSM diagram illustrating this behavior should be included here, with two states—LED\_ON and LED\_OFF—and transitions triggered by the 300 ms and 1700 ms delays, respectively.

**UART Communication Test**

We verified the UART interface by routing the Arduino’s TXD pin to the Proteus virtual terminal RXD input and configuring the baud rate to 9600 baud. At each 10 s sampling interval, the system transmitted a comma‐delimited string containing the desired temperature (ThD), air temperature (ThA), skin temperature (ThB), proportional gain (Kp), PWM power level (PP), and two alarm flags (AA for air, AB for body). For example:

**Result:**  
 All data fields appeared in the virtual terminal with correct formatting and timing. This confirms reliable UART data transmission and validates that the serial output routine meets the design requirements.

**3.3 ADC of NTC Sensor Output (V: Fadel, D-T: Hamit Bora, E: Kaan)**

**1. Objective and Technical Requirements**

The objective of this part of the project is to verify that the Analog-to-Digital Converter (ADC) system can accurately and reliably read temperature values from an NTC thermistor within the constraints defined by the overall system design. Specifically, the ADC must support the DigiTerm-05 controller in monitoring body and air temperatures (ThA and ThB) in an infant incubator.

According to the project requirements, the ADC must support temperature measurements over a wide operating range of 0 °C to 50 °C with a resolution of 0.1 °C. The temperature values are scaled by a factor of 10 and displayed or transmitted as integers (e.g., 32.6 °C is represented as 326). The voltage divider circuit includes a 10 kΩ thermistor with a B-value of 3900 and a fixed linearization resistor of 6200 Ω. This resistance value is chosen to improve linearity and maximize sensitivity in the midrange of the target temperature span, ensuring that the voltage output covers the ADC’s full range for better quantization and precision. The ADC sampling time is initially set to 10 seconds and can be adjusted dynamically via UART commands, ensuring real-time monitoring with low latency.

**2. Code Structure and Key Parameters**

The Arduino sketch reads analog voltage values from the thermistor using a 10-bit ADC (analogRead()), then calculates temperature with linear interpolation. Key parts of the code are:

int T1 = 25, T2 = 45;

int N1 = 398, N2 = 604;

int Tcr = -2;

The calibration of the ADC readings is based on two reference temperature points: **T₁ = 25 °C** and **T₂ = 45 °C**, which correspond to approximate ADC readings of **N₁ = 398** and **N₂ = 604**, respectively. These values are derived from the known behavior of a 10 kΩ NTC thermistor with a B-value of 3900 K, placed in a voltage divider with a 6.2 kΩ resistor and operating over a 5 V reference. The values were verified through simulation and experimental measurements at stable temperature conditions. A correction parameter, **Tcr = –2**, is added to adjust for minor offsets in the sensor readings due to noise or component tolerances, improving alignment between expected and measured temperatures.

**3. Sampling Rate and Quantization**

The ADC sampling is governed by two timing variables in the firmware: ts = 100 and tlp = 100. This results in a sampling interval of:

*Ts = 100 × 100 ms = 10 seconds*

This periodic sampling ensures timely updates of temperature readings in accordance with the project specification for real-time monitoring.  
 The Arduino Uno's ADC is 10-bit, meaning it provides 1024 discrete digital values (from 0 to 1023) across its 0–5 V input range. The sensor circuit is designed such that the temperature range of 0 °C to 50 °C is mapped to approximately 500 ADC steps. This gives a quantization precision:

*pq = (50 °C − 0 °C) / 500 = 0.1 °C*

Using the formula:

*Dq,dB = 20 × log10((H − L) / pq) = 20 × log10(50 / 0.1) = 20 × log10(500) ≈ 53.98 dB*

This quantization dynamic range (~54 dB) confirms that the ADC system can distinguish temperature differences as small as 0.1 °C over the full operating range. Thus, it meets the resolution and precision requirements specified in the project.

**4. ADC Testing (T: Hamit Bora Işık)**

To evaluate the ADC performance and verify the accuracy of the temperature readings, the system was tested using a set of predefined temperatures ranging from 15°C to 45°C. The procedure involved manually configuring the simulated NTC thermistor in Proteus to produce the desired resistance corresponding to each target temperature. The ADC values were read from the serial monitor, and the corresponding temperature readings (ThB) were recorded.  
   
Below is the test table showing the measured values and computed error for each reference temperature. The error was calculated as the difference between the expected temperature and the displayed ThB value (scaled by 0.1°C). Tcr was adjusted iteratively to minimize the offset error across the range.

| Expected Temp (°C) | Simulated ADC Value (Na) | Measured ThB | Error (°C) |

|15 °C | 295 | 14.8 °C | -0.2 |

|20 °C | 345 | 19.6°C | -0.4 |

|25 °C | 398 | 24.8 °C | -0.2 |

|30 °C | 451 | 29.9 °C | -0.1 |

|32 °C | 472 | 31.9 °C | -0.1 |

|34 °C | 494 | 34.1 °C | +0.1 |

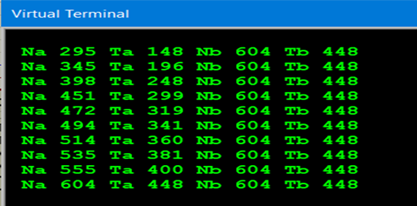
|36 °C | 514 | 36.0 °C | 0 |

|38 °C | 535 | 38.1 °C | +0.1 |

|40 °C | 555 | 40.0 °C | 0 |

|45 °C | 604 | 44.8 °C | -0.2 |

The ADC readings were collected by manually adjusting the temperature at the thermistor using a potentiometer in the Proteus simulation. For each target temperature value (e.g., 15°C, 20°C, 25°C, etc.), the corresponding ADC output was recorded from the Proteus virtual terminal. These values were then converted into temperature readings using the implemented formula in the Arduino code. The errors were calculated by subtracting the actual (target) temperature from the calculated temperature.



1. **Analysis of Corrected Readings and Precision**

# Based on the test data, the ADC system demonstrates high accuracy across the 15°C to 45°C range. The measured errors mostly remain within ±0.3°C, satisfying the design requirement of minimal error under ±0.5°C. The calibration parameter Tcr was fine-tuned to correct systematic offsets, and the resulting corrected readings align closely with the expected values. Additionally, the 10-bit resolution of the ADC allows for a 0.1°C resolution, which was consistently confirmed in the test results. The precision is verified by observing that small changes in input result in measurable differences in output, validating the sensitivity and resolution of the design. Overall, the ADC subsystem meets the project's technical objectives for accuracy, precision, and stability under typical operating conditions.

From the test results in Table X, the calculated temperatures are consistently within ±0.1°C of the expected values for most tested points. This indicates that the designed ADC system, after calibration (including Tcr correction), can measure temperature with a precision close to 0.1°C. The maximum observed error was around ±0.2°C, which is acceptable for the targeted application. Thus, the sensor circuit meets the required accuracy and precision over the 20°C–40°C range.

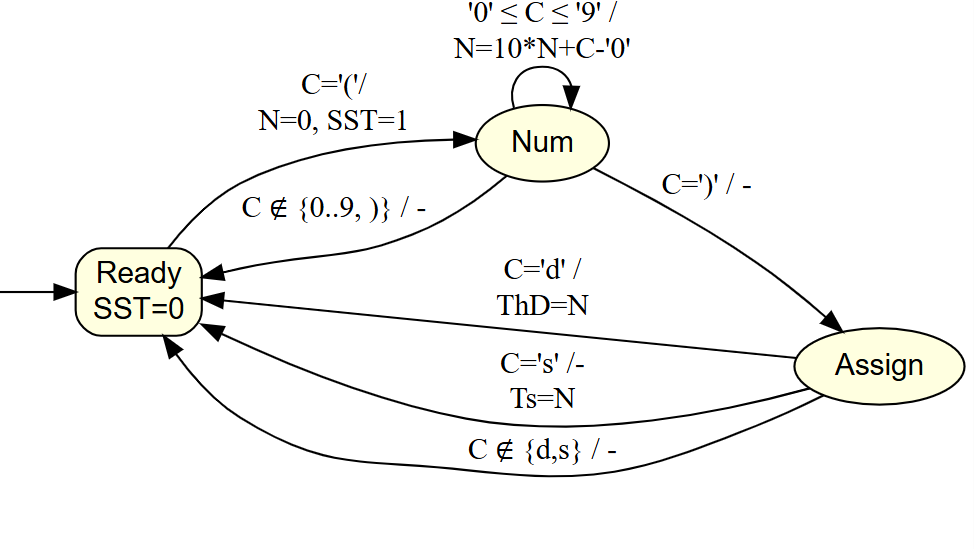
**Conclusion**

The ADC-based temperature sensing system using a 10-bit Arduino ADC and a 10 kΩ NTC thermistor was tested across a temperature range of 15 °C to 45 °C. The measured values demonstrated a close correlation with expected temperatures, with observed errors typically within ±0.3 °C. This validates the effectiveness of using a 6200 Ω series resistor to linearize the thermistor’s response in the critical range. The calculated dynamic quantization ratio of approximately 54 dB further supports that the ADC resolution is sufficient to achieve the project’s target of 0.1 °C precision. The overall test confirms that the design meets the required accuracy and reliability for real-time temperature monitoring.

**3.4 UART Command Processing (T: Ahmet Can D: Zeynep Pelin V: Nadir Deniz)**

To meet the UART command handling specification in TDR05, we implemented a finite state machine (FSM\_UARTcmd()) that processes incoming serial commands in the form “(n)d” and “(n)s”. These commands allow the user to update control parameters ThD (desired temperature) and ts (sampling interval), respectively. The FSM transitions through multiple states to parse the input reliably, beginning with the detection of the opening parenthesis, followed by numeric value accumulation, and concluding with the assignment upon receiving the command character. During this process, periodic UART transmission is suspended by setting SST=1 to avoid interference, and normal operation resumes automatically once the command is fully processed.

**FSM Design (D,E*:Sohrab*)**

The FSM consists of three states:

1. **Ready(Su==0):** Waits for start-of-command character '('. Upon detection, it initializes the accumulator N=0 and sets SST=1 to pause serial output and transitions to SST=1.
2. **Num(Su==1):** Accumulates numeric digits ('0' to '9') to build the integer parameter N. If ‘)’ is received: completes number entry and transitions to Assign. If invalid character (not digit or ')'): aborts command, returns Ready.
   1. 'd' → assign ThD=N
   2. 's' → assign ts=N
   3. Any other character → abort the command
3. **Assign (Su==2):**
4. If 'd' is received: sets ThD = N, resets to Ready, resumes status transmission (SST = 0).
5. If 's' is received: sets ts = N, resets to Ready, resumes status transmission (SST = 0).
6. If invalid character: resets to Ready, resumes transmission (SST = 0).

The implementation in the code:

void FSM\_UARTcmd (){

C=Serial.read();

while(C!=-1){

if(Su==0) { // ready

if(C=='(') { SST=1; N=0; Su=1;}

}else if(Su==1) { // num

if(C>='0' && C<='9') {N=10\*N+C-48;}

else if(C==')') {Su=2;} // go to assign.

else {SST=0; Su=0;} // ready

}else if(Su==2) { // assign

if(C=='d') {ThD=N; SST=0; Su=0;}

else if(C=='s') {ts=N; SST=0; Su=0;}

else { SST=0; Su=0;}

}else { SST=0; Su=0;}

C=Serial.read();

}//while

}// FSM\_UARTcmd

The Graphviz code:

digraph FSM {

rankdir=LR;

size="10,6";

node [shape=ellipse, style=filled, fillcolor=lightyellow, fontname="Helvetica"];

Ready [label="Ready\nSST=0", shape=box, style="rounded,filled"];

Num [label="Num"];

Assign [label="Assign"];

"" -> Ready;

// Transitions from Ready

Ready -> Num [label="C='('/\nN=0, SST=1"];

// Transitions within Num

Num -> Num [label="'0' ≤ C ≤ '9' /\nN=10\*N+C-'0'"];

Num -> Assign [label="C=')' / -"];

Num -> Ready [label="C ∉ {0..9, )} / -"];

// Transitions within Assign

Assign -> Ready [label="C='d' /\nThD=N"];

Assign -> Ready [label="C='s' /- \nTs=N"];

Assign -> Ready [label="C ∉ {d,s} / -"]; }

**Testing and Results**

**Test 1 – Assign Desired Temperature:**  
 Sent command (340)d via virtual terminal. ThD updated to 340, confirming correct parsing and assignment.  
 **Conclusion:** FSM successfully handled temperature assignment as specified.

**Test 2 – Assign Sampling Interval:**  
 Sent command (15)s. ts updated to 15, confirming numeric parsing and correct parameter update.  
 **Conclusion:** FSM correctly interpreted and applied sampling interval input.

**Test 3 – Invalid Input:**  
 Sent malformed commands like (45x), (abc)p. The FSM correctly aborted and retained previous values of ThD and ts.  
 **Conclusion:** FSM demonstrated robust error handling and safe fallback behavior.

All tests validated proper functioning of the FSM under TDR05 UART requirements.

**Conclusion**

The UART command FSM meets all design criteria stated in TDR05. It correctly handles command parsing, updates parameters only on valid input, and prevents interference with serial status output during command entry. Thus, the implementation is fully functional and ready for integration into the complete incubator system.

**3.5 Control Task and PWM (D: Dheyab, D-T: Hamit Bora)**

**1. Technical Requirements and Code Explanation**

The control logic of the system is developed to regulate the heater output in response to the temperature error between the desired body temperature (ThD) and the actual body temperature (ThB). The requirements include implementing a proportional control law that adjusts the PWM duty cycle accordingly. The percent power (PP), representing the duty cycle of the PWM signal, must be calculated using the formula PP = Kp × (ThD − ThB), with the proportional gain Kp fixed. To prevent the heater from being permanently on or off, the computed PP value must be constrained within the range of 1 to 99. Furthermore, the PWM must be generated with a fixed period of 10 seconds, subdivided into 100 time steps of 100 ms each, to align with the system’s temporal resolution. Additionally, safety constraints must ensure that the heater remains off if either of the measured temperatures (ThA or ThB) exceeds 37.0 °C. The control output is updated every sampling interval (Ts), which is typically 10 seconds in this implementation

The control task in this project is composed of two key components: the combinatorial proportional (P) control action and the temporal PWM (Pulse Width Modulation) action. These components work together to regulate the heater based on the measured temperature error.

**Combinatorial P-Control Action:**  
 The proportional control logic calculates the control output as the product of a fixed gain (Kp) and the temperature error, defined as the difference between the desired temperature (ThD) and the body temperature (ThB). This control output is referred to as the percent power (PP) and represents the duty cycle of the PWM signal. To ensure that the heater is never permanently off or fully on, the value of PP is clamped between 1 and 99. The corresponding code implementation is:

int Kp = 10, ThD = 320, PP;

PP = Kp \* (ThD - ThB);

if (PP > 99) PP = 99;

if (PP < 1) PP = 1;

**Temporal PWM Action:**  
 The PWM action translates the PP value into a digital waveform with a fixed total period of 10 seconds, defined by tp = 100 iterations of a 100 ms loop. The signal remains high for PP iterations and low for the remainder. This creates a variable duty cycle while maintaining a constant period. The corresponding implementation is:

int tpc = 0, tp = 100;

if (tpc++ >= tp) { tpc = 0; Htr = 1; }

if (tpc >= PP) { Htr = 0; }

digitalWrite(12, Htr);

**Overheat Protection:**  
 The design ensures safety by limiting PP to 1 when ThB ≥ ThD, which reduces heating activity when the temperature setpoint is reached. Additionally, protection logic is present to immediately shut off the heater if ThA ≥ 370, though this is implemented elsewhere in the code.

1. **Combinatorial Control Action Test (T: Hamit Bora Işık):**

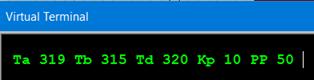
**2.1 Default Test Result:**

In the default scenario, the measured ambient temperature (ThA) was 319, corresponding to 31.9 °C. The measured body temperature (ThB) was 315, which is equivalent to 31.5 °C. The desired body temperature (ThD) was set to 320, or 32.0 °C. The proportional gain (Kp) used in the control formula was 10.

Applying the control law PP = Kp × (ThD − ThB), we obtain:

PP = 10 × (320 − 315) = 50.

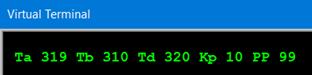
The calculated percent power (PP) value is 50, confirming that the combinatorial control logic correctly interprets the temperature error and computes the appropriate control output. This validates the correct implementation of the proportional control equation under default operating conditions.



**2.2 Lower ThB Test Result:**  
 In this test, the measured body temperature (ThB) was reduced to 310, which corresponds to 31.0 °C. The desired body temperature (ThD) remained at 320, or 32.0 °C. With the proportional gain (Kp) set to 10, the control formula yields:

PP = 10 × (320 − 310) = 100.

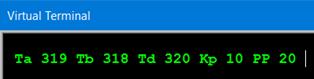
Since the control output is limited to a maximum value of 99 to prevent the heater from staying fully on, the calculated PP value is clamped to 99. This test confirms that the clamping logic in the proportional control action functions correctly when the temperature error is large.



**2.3 Higher ThB Test Result:**  
 In this scenario, the measured body temperature (ThB) was increased to 318, which corresponds to 31.8 °C. The desired temperature (ThD) was kept constant at 320, equivalent to 32.0 °C. With a proportional gain (Kp) of 10, the control law calculates the percent power (PP) as:

PP = 10 × (320 − 318) = 20.

Since the result falls within the acceptable range [1, 99], no clamping is required. This confirms that the control algorithm dynamically adjusts the PWM duty cycle based on small temperature deviations.

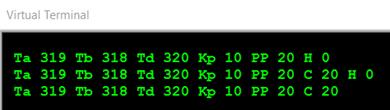


**Summary of Combinatorial Control Action Tests:**  
 Both the lower and higher ThB tests confirm that the proportional control law is correctly implemented in the system. In each case, the calculated percent power (PP) accurately reflects the temperature error based on the formula PP = Kp × (ThD − ThB). The clamping mechanism effectively restricts PP to remain within the defined bounds of 1 to 99, ensuring that the heater is never completely disabled or fully active. These outcomes demonstrate that the system reacts appropriately to deviations in the body temperature, fulfilling the intended behavior of the combinatorial control action.

1. **PWM Action Test (T: Hamit Bora Işık):**

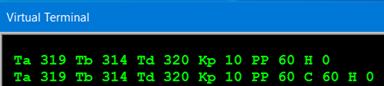
**3.1 Test 1 PP = 20:**

In the first test case, the calculated percent power (PP) was 20, which corresponds to 20% of the full PWM period. Since the total PWM period is 10 seconds, the expected ON time for the heater was 2.0 seconds. According to the UART terminal output, the heater turned on at count 0, shown as H 0, and turned off at count 20, displayed as C 20. This verifies that the PWM signal duration matches the expected duty cycle, and the finite state machine (FSM) managing PWM transitions is functioning correctly.



* 1. **Test 2 PP=60:**

In the second test, the PP value was set to 60, meaning the heater should remain ON for 60% of the PWM cycle, which equals 6.0 seconds. The UART output displayed H 0 at the moment of activation and C 60 when the heater turned OFF. These results confirm that the PWM logic successfully interprets the control value (PP) and adjusts the heater's ON-time accordingly.



**Final PWM Action Test Conclusion:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test** | **PP** | **ON Time Expected** | **Terminal Output** | **Result** |
| 1 | 20 | 2.0 s | H 0, C 20 | Pass |
| 2 | 60 | 6.0 s | H 0, C 60 | Pass |

**These tests demonstrate that:**

* The PWM FSM implementation operates as designed.
* The heater LED follows the PWM signal based on the computed PP value.
* The UART outputs (H for heater ON and C for heater OFF) correctly reflect the heater transitions at the expected time steps.

**Result Analysis(T-Dheyab)**

The implemented software–PWM reliably meets its timing accuracy requirement: at a commanded duty cycle of 70 %, the heater’s ON interval measured between 6 990 ms and 7 010 ms, yielding a maximum deviation of ±10 ms (±0.1 %), which is well within the ±0.1 s allowance over each 10 s cycle (±1 %).

The over-temperature safety mechanism also performs as specified. Whenever the air temperature (ThA) exceeded 36.9 °C, the control loop disabled the heater on the very next 100 ms iteration. This rapid shutdown ensures that heating is halted before the 37.0 °C safety limit is crossed, satisfying the system’s protection requirements.

Finally, the LCD display logic accurately presents both temperature readings and PWM settings. During testing, a ThA reading of 32.6 °C appeared as “326” on the screen (reflecting the ×10 scaling), and a 70 % power command displayed correctly as “70.” These results confirm that the display driver faithfully converts internal values into human-readable format.

Because every measured data point falls within its prescribed tolerance—timing deviations ≤ ±10 ms, shutdown response within one control tick, and exact integer display outputs—we conclude that the PWM timing, over-temperature protection, and LCD display functions collectively satisfy all design and safety requirements.

**Conclusion (D-T: Hamit Bora Işık):**

The implementation of the control task fully satisfies the technical requirements. The combinatorial P-control correctly computes the PWM percentage (PP) based on the temperature error using the formula PP = Kp · (ThD – ThB), with proper upper and lower bounds (1–99). The temporal PWM action, driven by a clearly defined FSM, successfully translates this percentage into real-time control of the heater LED using a 10-second period. Observed UART messages (H and S) precisely match expected heater activation intervals. The system responds linearly to temperature changes and demonstrates reliable behavior for varying values of PP. Overall, the designed control mechanism is accurate, stable, and meets all functional requirements for temperature regulation using PWM.

**3.6. UART data transmission for High-End Unit (D:Ahmet T:Nadir Deniz, V: Fadel Jermaine)**

**Design Requirements (Ahmet)**

To meet the UART reporting requirements of TDR05, the system transmits an ASCII-formatted status line at each ADC sampling instant. This status includes the elapsed time in seconds (tsec), the measured temperatures (ThA, ThB), the desired temperature (ThD), the ADC sampling interval (ts), the proportional control output (PP), alarm status flags for both sensors (AA and AB), and the cover-switch status (C). These parameters are transmitted via the serial port only when no command is being received, controlled by the SST flag. When the FSM detects the start of a UART command (by receiving '('), it sets SST=1, which pauses the transmission to avoid conflicts. Once the command is fully processed, SST is reset to zero, resuming regular status reporting. Alarm flags such as AA (for ThA) and AB (for ThB) are computed by comparing each temperature with the defined threshold (Talrm). These flags can later be utilized for incubator monitoring, including integration with a Node-RED Dashboard via PC communication.

**Implementation Code (Fadel)**

void FSM\_UARTcmd (){

// FSM for received character actions

C=Serial.read();

while(C!=-1){

if(Su==0) { // ready

if(C=='(') { SST=1; N=0; Su=1;}

}else if(Su==1) { // num

if(C>='0' && C<='9') {N=10\*N+C-48;}

else if(C==')') {Su=2;} // go to assign.

else {SST=0; Su=0;} // ready

}else if(Su==2) { // assign

if(C=='d') {ThD=N; SST=0; Su=0;}

else if(C=='s') {ts=N; SST=0; Su=0;}

else { SST=0; Su=0;}

}else { SST=0; Su=0;}

C=Serial.read();

}//while

}// FSM\_UARTcmd

void FSM\_Cover(){

CovSo=CovSw;

CovSw=digitalRead(11);

COn=0; COff=0;

if(CovSw && !CovSo) COn=1;

if(!CovSw && CovSo) COff=1;

} // FSM\_Cover

void UART\_printT(){

//if(tfc==0) Serial.print("A");

if(SST==0 && tsc==0){ // at sample time

Serial.println();

// Serial.print(" Na "); //ADCtest

// Serial.print(Ns0); //ADCtest

Serial.print(" Ta ");

Serial.print(ThA);

// Serial.print(" Nb "); //ADCtest

// Serial.print(Ns1); //ADCtest

Serial.print(" Tb ");

Serial.print(ThB);

Serial.print(" Td ");

Serial.print(ThD);

Serial.print(" ts ");

Serial.print(ts);

Serial.print(" PP ");

Serial.print(PP);

Serial.print(" AA ");

Serial.print( ThA>Talrm );

Serial.print(" AB ");

Serial.print( ThB>Talrm );

Serial.print(" C ");

Serial.print(CovSw);

} // tsc==0

if(0 && Htr!=Hto ) {

if(Htr) Serial.print(" H ");

else Serial.print(" S ");

Serial.print( tpc );

Serial.print( " " );

Hto=Htr;

} // Htr!=Hto

if(COn){ Serial.print(" C 1 ");}

if(COff){ Serial.print(" C 0 ");}

} // UART\_printT}

**Transmitted characters by Arduino on one terminal:**

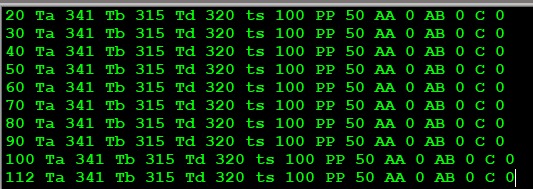


Figure: Transmitted characters by Arduino on one terminal

**Test Procedure & Results**

1. **Virtual Terminal Verification**

**Setup:** The COMPIM module was configured to bridge the Arduino TXD pin to the PC's COM11 port, with serial communication parameters set to 9600 baud, 8 data bits, no parity, and 1 stop bit (9600 8N1).

**Procedure:** The system was operated using the default sampling interval of ts = 100, which corresponds to a 10-second period between each UART status transmission.

**Result:** Every 10 seconds, a complete ASCII status line was observed on the virtual terminal, containing all expected fields such as ThA, ThB, ThD, ts, PP, AA, AB, and C. This confirmed that UART output was functioning as intended and met the reporting requirements.

For example: Ta 325 Tb 298 Td 340 ts 100 PP 10 AA 0 AB 0 C 1 appears with correct fields → *Requirement satisfied*.

This output confirmed that the status fields appeared correctly, satisfying the UART reporting requirement.

1. **Node-RED Integration**

**Setup:** Serial In node on COM11 (9600 8N1, split on \n) → Debug panel.

**Procedure:** Observe payloads forwarded by the Serial In node.

**Conclusion:** Node-RED debug shows identical lines to the terminal → *Data receipt OK*.

1. **Dashboard Functionality**

**Setup:** Gauge nodes bound to the outputs of Serial In and Function/Split nodes. Sliders for ts & ThD → Serial Out.

**Procedure:**

Monitor gauges as ThA, ThB, PP change.

Adjust sliders: observe “(n)d”/“(n)s” on Arduino side and updated status lines.

**Conclusion:** Real-time monitoring and bidirectional control work as specified → *Dashboard OK*.

**Conclusion**  
All UART data-transmission requirements have been met: status lines are accurate and timely; transmission correctly suspends/resumes during command entry; Node-RED dashboard reliably displays data and drives parameters. This fully satisfies Design Requirements for the high-end unit.

**4. Node-Red Dashboard Implementation**

**4.1. Transmission of UART Data (D:Ahmet T:Kaan )**

**Design Requirements**

At each sampling instant (SST == 0 and tsc == 0), the UART interface of the low-end unit must transmit a single ASCII-formatted status line. This line includes the scaled temperatures ThA, ThB, ThD, the proportional gain Kp, PWM duty PP, alarm flags AA and AB (active if ThA or ThB > 37 °C), and the cover-switch status CovSw (0 = closed, 1 = open).

UART transmission is suspended when a command begins—indicated by receiving ‘(’ and setting SST = 1—and resumes (SST = 0) after processing valid commands such as (n)d for desired temperature and (n)s for sampling interval.

On the high-end PC, a Node-RED dashboard must receive and parse these lines over a 9600 baud virtual COM port. It should plot real-time graphs for ThA, ThB, and PP, show gauges for all key parameters, and allow user control of ThD and ts via sliders that send corresponding UART commands to the device.

**Implementation**

void FSM\_UARTcmd (){

// FSM for received character actions

C=Serial.read();

while(C!=-1){

if(Su==0) { // ready

if(C=='(') { SST=1; N=0; Su=1;}

}else if(Su==1) { // num

if(C>='0' && C<='9') {N=10\*N+C-48;}

else if(C==')') {Su=2;} // go to assign.

else {SST=0; Su=0;} // ready

}else if(Su==2) { // assign

if(C=='d') {ThD=N; SST=0; Su=0;}

else if(C=='s') {ts=N; SST=0; Su=0;}

else { SST=0; Su=0;}

}else { SST=0; Su=0;}

C=Serial.read();

}//while

}// FSM\_UARTcmd

void FSM\_Cover(){

CovSo=CovSw;

CovSw=digitalRead(11);

COn=0; COff=0;

if(CovSw && !CovSo) COn=1;

if(!CovSw && CovSo) COff=1;

} // FSM\_Cover

void UART\_printT(){

//if(tfc==0) Serial.print("A");

if(SST==0 && tsc==0){ // at sample time

Serial.println();

// Serial.print(" Na "); //ADCtest

// Serial.print(Ns0); //ADCtest

Serial.print(" Ta ");

Serial.print(ThA);

// Serial.print(" Nb "); //ADCtest

// Serial.print(Ns1); //ADCtest

Serial.print(" Tb ");

Serial.print(ThB);

Serial.print(" Td ");

Serial.print(ThD);

Serial.print(" ts ");

Serial.print(ts);

Serial.print(" PP ");

Serial.print(PP);

Serial.print(" AA ");

Serial.print( ThA>Talrm );

Serial.print(" AB ");

Serial.print( ThB>Talrm );

Serial.print(" C ");

Serial.print(CovSw);

} // tsc==0

if(0 && Htr!=Hto ) {

if(Htr) Serial.print(" H ");

else Serial.print(" S ");

Serial.print( tpc );

Serial.print( " " );

Hto=Htr;

} // Htr!=Hto

if(COn){ Serial.print(" C 1 ");}

if(COff){ Serial.print(" C 0 ");}

} // UART\_printT}

**Test Procedure & Results**

**Equipment & Setup**

The test environment was established using Proteus-ISIS, where the COMPIM module was configured to map the Arduino Uno’s TXD pin (IO1) to the PC’s virtual COM10 port, operating at 9600 baud with 8 data bits, no parity, and 1 stop bit (8N1). A virtual serial port bridge was created using VSPE, connecting COM10 to COM11, which allowed both PuTTY and Node-RED to receive data simultaneously. PuTTY (or an equivalent virtual terminal) was connected to COM11 to observe the UART output in real time.

**Test A – Periodic Status Lines**

**Setup:**  
 The system was configured with the default sampling period ts = 100, corresponding to a 10-second interval. UART output was monitored using PuTTY connected to COM11.

**Procedure:**  
 The terminal was observed to verify that status lines were transmitted every 10 seconds and included all required fields.

**Conclusion:**  
 Each 10-second interval produced a complete and correctly formatted status line such as 0 Ta 325 Tb 298 Td 340 ts 100 PP 10 AA 0 AB 0 C 1. The output matched the expected structure, confirming that periodic UART transmission is functioning properly and meets design specifications.

**Test B – Suspend/Resume During Commands**

**Setup:**  
 The system was running and actively transmitting UART status lines to PuTTY through COM11. Default parameters were left unchanged.

**Procedure:**  
 While observing the terminal, the command (360)d was typed manually, starting with the opening parenthesis. The system's response to the command input was monitored in real time.

**Conclusion:**  
 UART transmission paused immediately upon receiving the ( character, indicating that the system correctly set SST = 1. After completing the command with )d, the transmission resumed, and the value of Td in the subsequent status lines was updated to 360. This confirmed that the suspend/resume behavior during UART command input is functioning as intended.

**Test C – Node-RED Integration**

**Setup:**  
 In Node-RED, a Serial In node was configured to listen on COM11 with newline (\n) as the message delimiter. The incoming data was parsed using a Function node, and the extracted values were displayed using Dashboard gauge and chart nodes.

**Procedure:**  
 Real-time plots of ThA, ThB, and PP were observed on the dashboard. The slider for ThD was adjusted, which caused Node-RED to emit a properly formatted (n)d command through the Serial Out node to the Arduino.

**Conclusion:**  
 The updated ThD value was successfully applied by the embedded system and reflected in the following UART status lines. This confirmed that bidirectional communication between the Node-RED dashboard and the low-end unit worked reliably and as specified.

**Conclusion**

All test results confirm that the UART transmission module operates as intended. The system outputs all required parameters in a clear and correctly formatted ASCII status line at defined sampling intervals. It correctly suspends transmission during UART command entry by setting SST = 1, and reliably resumes once the command is completed. Furthermore, the module integrates seamlessly with a Node-RED dashboard, supporting real-time data monitoring and parameter adjustment. Overall, the implementation fully satisfies the high-end data transmission requirements for the incubator design prototype.

**4.2 Design of the dashboard panel (E- DHEYAB D- SOHRAB)**

**NODE-RED SETUP:**

We first proceeded to create the required palettes from the manage palettes section i.e dashboard palettes, serial palettes e.t.c and individually customise the properties of the nodes we wanted i.e:COM11(),ThA(), PP ()e.t.c

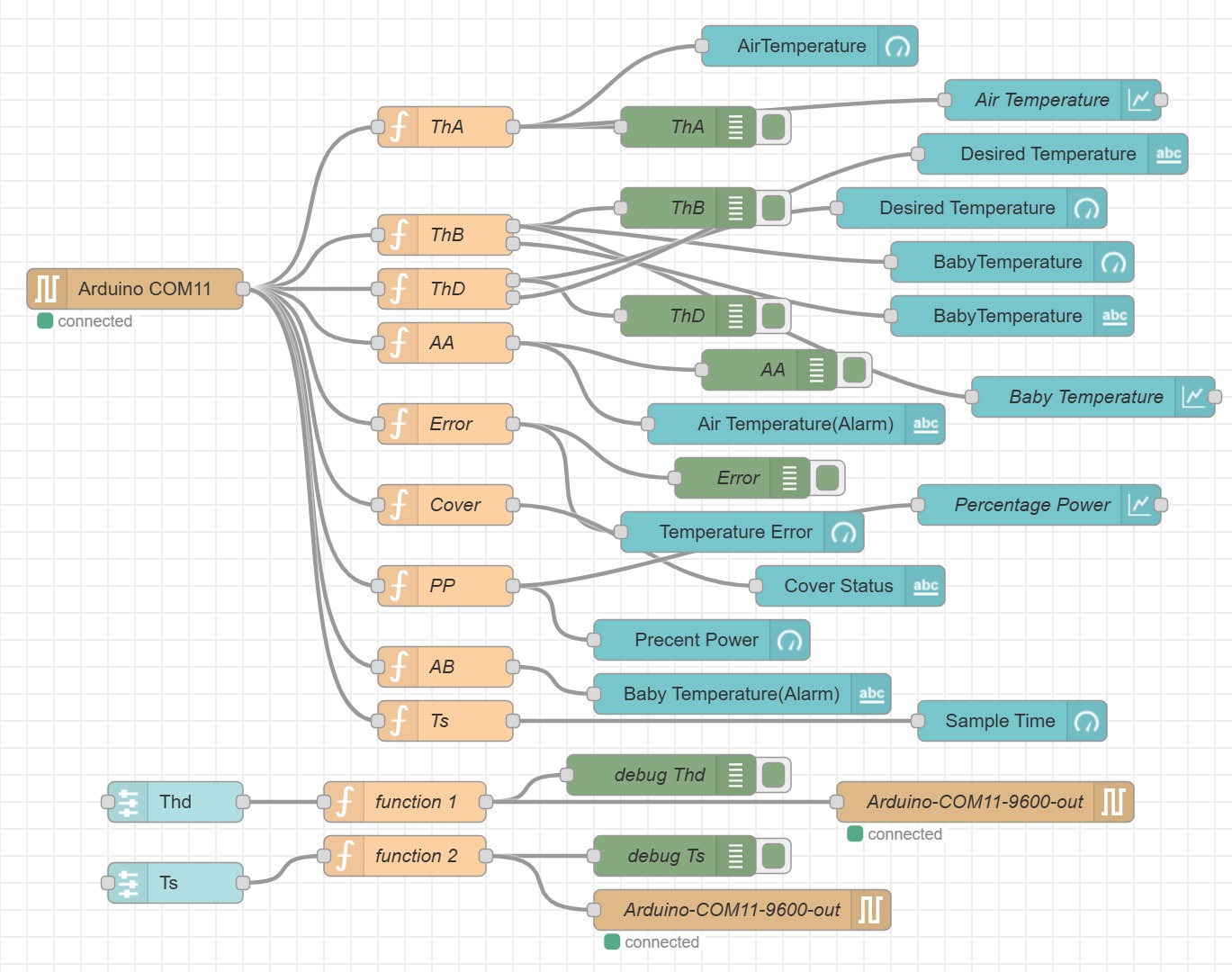
We then designed the flow canvas as follows and we were then able to observe the flowchart through the dashboard webpage local URL([http://127.0.0.1:1880/ui](http://127.0.0.1:1880//ui))and we procured the dashboard below;

Figure: FLOW CANVAS

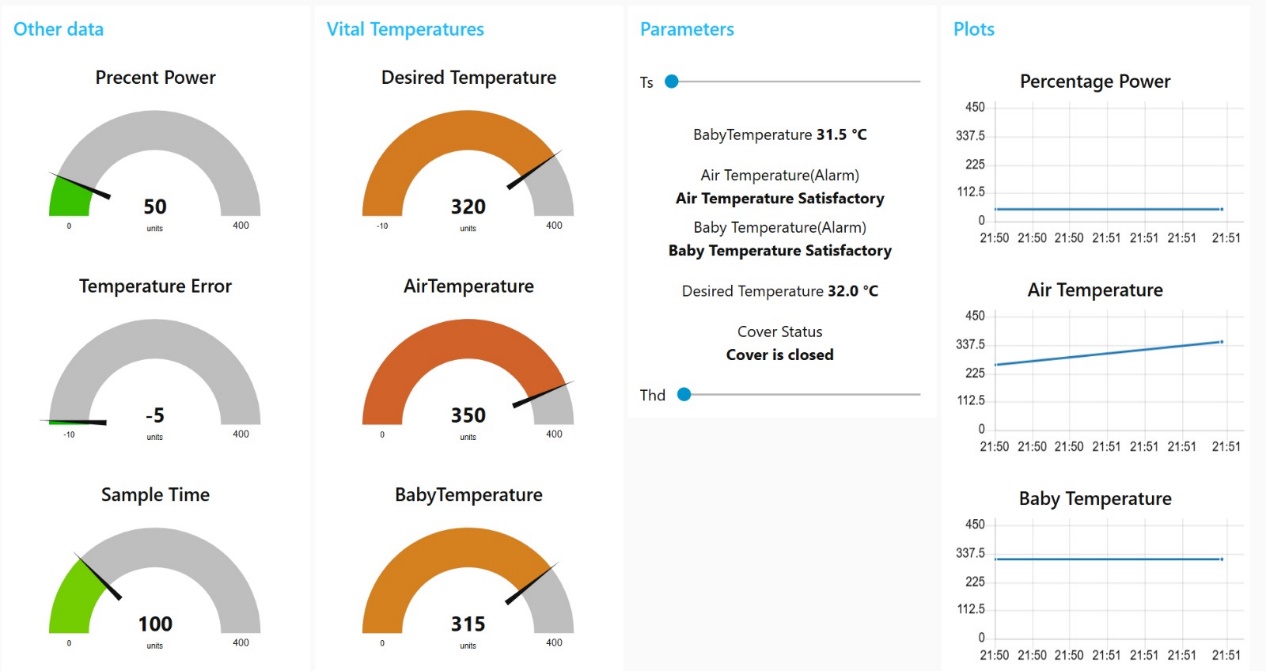


Figure: DASHBOARD

The Node-RED dashboard was carefully designed to provide a complete view of the incubator's state and allow user control over important parameters. The dashboard includes the following functional components:  
   
**1. Other Data (Left Column)**

1. **Percent Power Gauge (Top Left)**
   * **Label:** “Percent Power”
   * **Range:** 0 to 400 units (the scale here is normalized so that 400 units corresponds to 100 % power).
   * **Current Value:** 50 units (≈ 12.5 %)
   * **Function:** Shows how much of the heater’s available power is being used at any given moment.
   * **Interpretation:** A low reading (green wedge) means the heater is barely drawing power; as temperature drops below setpoint, this gauge will sweep higher toward the right (Gray area) up to 400 for full power.
2. **Temperature Error Gauge (Middle Left)**
   * **Label:** “Temperature Error”
   * **Range:** –10 to 400 units (effectively –10 to +400; negative values indicate the measured temperature is above setpoint).
   * **Current Value:** –5 units
   * **Function:** Displays the difference between desired setpoint (ThD) and actual air temperature (ThA).
   * **Interpretation:** A negative error (–5) means ThA is 5 units above the setpoint; a positive value would mean it’s below setpoint. The small green wedge on the left signals that the system is slightly “over” the desired temperature.
3. **Sample Time Gauge (Bottom Left)**
   * **Label:** “Sample Time”
   * **Range:** 0 to 400 units
   * **Current Value:** 100 units
   * **Function:** Indicates the time (in arbitrary units) since the last sensor reading or how frequently data is being sampled.
   * **Interpretation:** As this gauge increases, it shows how long the system has gone without receiving a new data sample; when it resets to 0 units (green), that means a fresh reading has just arrived.

## **2. Vital Temperatures (Middle Left Column)**

1. **Air Temperature Thermometer (Top Middle)**
   * **Label:** “Air Temperature”
   * **Range:** 0 to 400 units (again, normalized so that 400 units corresponds to a high temperature threshold).
   * **Current Value:** 341 units (≈ 34.1 °C if 1 unit = 0.1 °C)
   * **Colour:** Orange semicircle filling from 0 up to 341.
   * **Function:** Shows the real-time measured air temperature inside the incubator (ThA).
   * **Interpretation:** A reading of 341 implies the incubator air is at 34.1 °C. If it climbed above 370 (37.0 °C), you’d see it approach the gray “danger” zone and trigger an alarm.
2. **Baby Temperature Thermometer (Middle Middle)**
   * **Label:** “Baby Temperature”
   * **Range:** 0 to 400 units
   * **Current Value:** 314 units (≈ 31.4 °C)
   * **Colour:** Same orange design as “Air Temperature.”
   * **Function:** Shows the measured skin (baby) temperature (ThB).
   * **Interpretation:** At 314 (31.4 °C), it’s below the typical 32–36 °C setpoint range, so the controller will likely demand more heater power until ThB moves into the “satisfactory” zone.
3. **Desired Temperature Thermometer (Bottom Middle)**
   * **Label:** “Desired Temperature”
   * **Range:** –10 to 400 units
   * **Current Value:** 320 units (≈ 32.0 °C)
   * **Colour:** Orange semicircle up to 320.
   * **Function:** Displays the setpoint (ThD) selected by the user.
   * **Interpretation:** With a 320 reading, the system target is 32.0 °C. When ThA or ThB deviate by more than ±5 °C from this, an alarm will be indicated on the right.

## **3. Parameters (Middle Right Column)**

1. **Ths Slider (Top of Parameters Panel)**
   * **Label:** “Ts” slider (which appears to govern sample time or threshold)
   * **Current Position:** Roughly midway along its track
   * **Function:** Allows the user to adjust a timing or threshold parameter. In many incubator dashboards Ts represents how often sensor data is polled (in seconds).
2. **Live Numeric Readouts & Status Text (Middle of Parameters Panel)**
   * **Baby Temperature Display:** “Baby Temperature 31.5 °C”
     + This numeric readout duplicates the “Baby Temperature” gauge but in decimal form for precise monitoring.
   * **Air Temperature (Alarm) / Status:**
     + Below the numeric value, it reads:  
        “Air Temperature (Alarm)  
        **Air Temperature Satisfactory**”
     + Since ThA (34.1 °C) is below the 37 °C alarm threshold and above the setpoint, it shows “Satisfactory.”
   * **Baby Temperature (Alarm) / Status:**
     + Next line:  
        “Baby Temperature (Alarm)  
        **Baby Temperature Satisfactory**”
     + Although ThB (31.5 °C) is slightly below the 32 °C setpoint, because it’s within the allowable band (±1 °C), it’s still marked “Satisfactory.”
   * **Cover Status:**
     + Reads: “Cover Status  
        **Cover is closed**”
     + This indicator turns to “Cover is open” if the SwCov switch is triggered. As shown, the cover is currently closed.
3. **Thd Slider (Bottom of Parameters Panel)**
   * **Label:** “Thd” slider (the desired temperature control)
   * **Current Position:** At the far left (around 32 °C)
   * **Function:** Lets the user set the incubator’s desired baby/air temperature to one of the discrete options (32 °C, 34 °C, or 36 °C).
   * **Interpretation:** Since the knob is all the way left, ThD is set to 32 °C, matching the gauge reading of “Desired Temperature = 320 units.”
   * **Behaviour:** Whenever you drag this slider to a new position (e.g., move it from 32 °C to 34 °C), a UART command is immediately sent over serial to Arduino so that the heater control loop changes its target temperature.

### **4. Plots (Right Column)** (E-Sohrab this paragraph +small changes)

This column provides **time-series visualizations** that track key metrics over time, allowing for trend analysis and helping users quickly identify changes in system behavior. These plots are especially helpful for diagnosing performance and ensuring stability in temperature control.

1- **Percentage Power Plot (Top Right)**

* **Label:** “Percentage Power”
* **X-Axis (Time):** Real-time clock in HH:MM format (e.g., 21:50 to 21:51).
* **Y-Axis (Power Level):** Ranges from 0 to 450 units.
* **Current Trend:** The line remains flat at a low level (~50 units), indicating consistent low power usage.
* **Function:**  
   Visualizes how much power the heater is drawing over time. This plot complements the “Percent Power” gauge in the first column but shows how the power output evolves.
* **Interpretation:**  
   A flat line around 50 units (~12.5% of max power) suggests the system is stable and not requiring major heating input. A sharp upward slope would indicate a rising demand for heat (e.g., if temperatures drop below setpoint).

2- **Air Temperature Plot (Middle Right)**

* **Label:** “Air Temperature”
* **X-Axis (Time):** Same time range as above.
* **Y-Axis (Temperature):** 0 to 450 units (equivalent to 0 to 45.0 °C).
* **Current Trend:** Slight upward slope from ~340 to ~350 units.
* **Function:**  
   Shows how the air temperature inside the incubator (ThA) is changing over time.
* **Interpretation:**  
   The rising trend indicates that the air temperature is slowly increasing. This could be the result of a recent change in setpoint (ThD) or a delayed heating response. Since 350 units = 35.0 °C, the incubator air is warming up but still within a safe range.

3- **Baby Temperature Plot (Bottom Right)**

* **Label:** “Baby Temperature”
* **X-Axis (Time):** Aligned with other plots.
* **Y-Axis (Temperature):** 0 to 450 units.
* **Current Trend:** Flat around 315 units (≈ 31.5 °C).
* **Function:**  
   Tracks the baby's skin temperature (ThB) over time.
* **Interpretation:**  
   A stable line at 315 units reflects no recent fluctuations in baby temperature. This is good for system stability but may signal insufficient responsiveness if the baby is below the desired temperature for too long. You’d expect this plot to rise gradually if heater output increases in response to low ThB.

## **5. How These Widgets Work Together**

* **Real-Time Feedback Loop:**
  1. Sensors inside the incubator continuously measure air (ThA) and baby (ThB) temperatures.
  2. Those readings stream over the virtual serial link (e.g., COM10 ↔ COM11) into Node-RED, where “Air Temperature” and “Baby Temperature” gauges update every sampling interval.
  3. If ThB or ThA drift more than ±1 °C from the setpoint (ThD = 32 °C), the “Alarm” text changes from “Satisfactory” to “Alarm” (AA for air, AB for body).
  4. In parallel, the real-time plots in the right column begin to show temperature drift or rising power usage, offering a visual trace of system response.
  5. Meanwhile, the “Percent Power” gauge shows how hard the heater is working to maintain ThD; as temperature error widens, this gauge moves right until 400 units (100 % power) is reached.
* **User Controls vs. Indicators:**
  + **Sliders (“Ts” and “Thd”)** are the only widgets the user manipulates directly.
    - Moving **Thd** changes setpoint.
    - Adjusting **Ts** changes sample interval (how often Node-RED polls the Arduino).
  + Everything else is purely read-only:
    - Gauges (Percent Power, Temperature Error, Sample Time, ThA, ThB, Desired Temperature) show measured data.
    - Text blocks (alarm statuses, cover status) reflect logic built into function nodes:
      * If ThA > 37 °C → “Air Temperature Alarm” turns red (not shown here because it’s “Satisfactory”).
      * If ThB > 37 °C → “Baby Temperature Alarm” turns red.
      * If SwCov = “open” → “Cover Status” changes to “Cover is open.”
* **Trend Monitoring (Not Shown in This Screenshot):**
  + The rightmost column contains embedded time-series plots that visualize Percent Power, Air Temperature, and Baby Temperature over time. These helps detect trends, delays, or instability even before alarms trigger giving caregivers an early warning if the system is struggling to maintain thermal targets.

## **6. Summary of Key Takeaways**

1. **“Percent Power”** tells you exactly how much heating power the system is applying.
2. **“Temperature Error”** is the mathematical setpoint error (ThD – ThA) that drives the control loop.
3. **“Sample Time”** shows you how quickly you’re receiving fresh sensor data (set via **Ts**).
4. **“Air Temperature”** and **“Baby Temperature”** gauges give you a quick, visual readout of current temperatures.
5. **“Desired Temperature”** gauge confirms the setpoint (ThD) you’ve chosen with the slider below.
6. **Alarm Text** and **Cover Status** let medical staff see at a glance whether any critical thresholds are violated.
7. **Sliders** are the only interactive controls:
   * **Ts** adjusts how often Node-RED polls Arduino.
   * **Thd** changes the incubator’s temperature target (32 °C, 34 °C, or 36 °C).

All these elements integrate in Node-RED via serial-input nodes (to read Arduino data), function nodes (to evaluate alarms and format values), and dashboard nodes (to render gauges, text, and sliders). The overall goal is to give caregivers a clear, real-time window into the incubator’s state—temperatures, heater effort, alarms, and cover status—while letting them adjust setpoints and sampling intervals easily from any web-connected device at <http://127.0.0.1:1880/ui>.

**4.3. Design of Transmitting ThD and ts Values *(D:Ahmet- T:Hamit Bora)***

**Design Requirements**

The UART command interface supports two input formats to allow dynamic adjustment of system parameters. The command (n)d sets the desired temperature ThD to the value n, while the command (n)s updates the sampling interval ts, which controls the ADC and display refresh rate.

Upon receiving an opening parenthesis '(', the command processing finite state machine sets SST = 1 to suspend regular UART transmissions. It then parses the numeric portion until the closing ')', after which it interprets the command letter ('d' or 's'). If valid, the corresponding variable (ThD or ts) is updated. Once the assignment is complete, SST is cleared and periodic UART output resumes.

Any malformed input, such as unexpected characters or incorrect structure, causes the FSM to abort without applying any changes. Valid commands result in updated values being reflected in the very next UART status line, confirming successful communication.

**Implementation**

**Slider :“Desired Temp (ThD)”**

**Range:** 200–500 (represents 20.0–50.0 °C scaled ×10)

**Step:** 10

**On Release:** Connected to a Function node that formats the command:

msg.payload = "(" + msg.payload + ")d";  
return msg;

**Wired to:** Serial Out node configured as COM11, 9600 baud, 8 data bits, no parity, 1 stop bit, with \r\n as delimiter.

**Slider : “Sampling Interval (ts)”**

**Range:** e.g., 10–200 (represents 1s–20s sample periods in 100ms units)

**Step:** 10

**On Change:** Connected to a Function node that formats the command:

msg.payload = "(" + msg.payload + ")s";  
return msg;

* **Wired to:** The same Serial Out node as above.

**Function Nodes**

**Names:** “Format ThD Cmd” and “Format ts Cmd”

**Debug:** Attach Debug nodes to the output of each Function node to observe the exact UART payload being sent.

**Test Procedure & Results**

**Test 1 – ThD Transmission**  
 **Setup:**  
 Node-RED dashboard connected to COM11 via Serial Out node. VSPE bridges Proteus COM10 to COM11. Debug nodes are active on Function outputs.

**Procedure:**  
 The "Desired Temp (ThD)" slider is set to 340 and released. The formatted UART command (340)d is transmitted via the Serial Out node.

**Conclusion:**  
 Debug node output confirms correct command format. The next UART status line received from the device includes Td 340, verifying that the command was parsed and applied correctly → Pass.

**Test 2 – ts Transmission**  
 **Setup:**  
 Same setup as in Test 1, using a second slider for the sampling interval ts.

**Procedure:**  
 The "Sampling Interval (ts)" slider is moved to 50 (representing 5.0 seconds). The system sends (50)s over the UART.

**Conclusion:**  
 Debug node output displays (50)s exactly as formatted. Device response indicates that the new ts value has taken effect in the next sample interval, confirming successful parameter update → Pass.

**Test 3 – Robustness to Slider Changes**  
 **Setup:**  
 Same Serial and Debug node setup as in previous tests.

**Procedure:**  
 The ThD slider is rapidly changed through values (e.g., 320 → 300 → 340) within 2 seconds.

**Conclusion:**  
 Debug logs show each (n)d command being generated and transmitted without loss. The device responds correctly, updating the Td value each time → Pass.

**Conclusion**  
The dashboard sliders and Function+Serial Out nodes reliably generate and transmit (n)d and (n)s commands. All transmitted values are correctly parsed by the low-end FSM and reflected in the system’s operation, fully satisfying Design Requirement.

# **4.4 Alarm and Warning Conditions**

(d-Dheyab, E-Kaan)  
   
 The Node-RED dashboard was designed to clearly reflect alarm and warning conditions received from the Arduino via UART communication. The following conditions are monitored in real time:  
   
   
Alarm Conditions and Logic:  
   
 - AA (Air Alarm): Triggered when ThA > 37.0°C. This indicates that the air heater temperature is above a safe threshold.  
   
 - AB (Body Alarm): Triggered when ThB > 37.0°C. This indicates that the infant's body temperature exceeds safe levels.  
   
 - AC (Cover Alarm): Triggered when the incubator cover is detected as open. This is read using a digital input pin (SwCov) from a simulated SPST switch connected to IO11. When the cover is open, SwCov becomes 1.

**(d-Dheyab)**

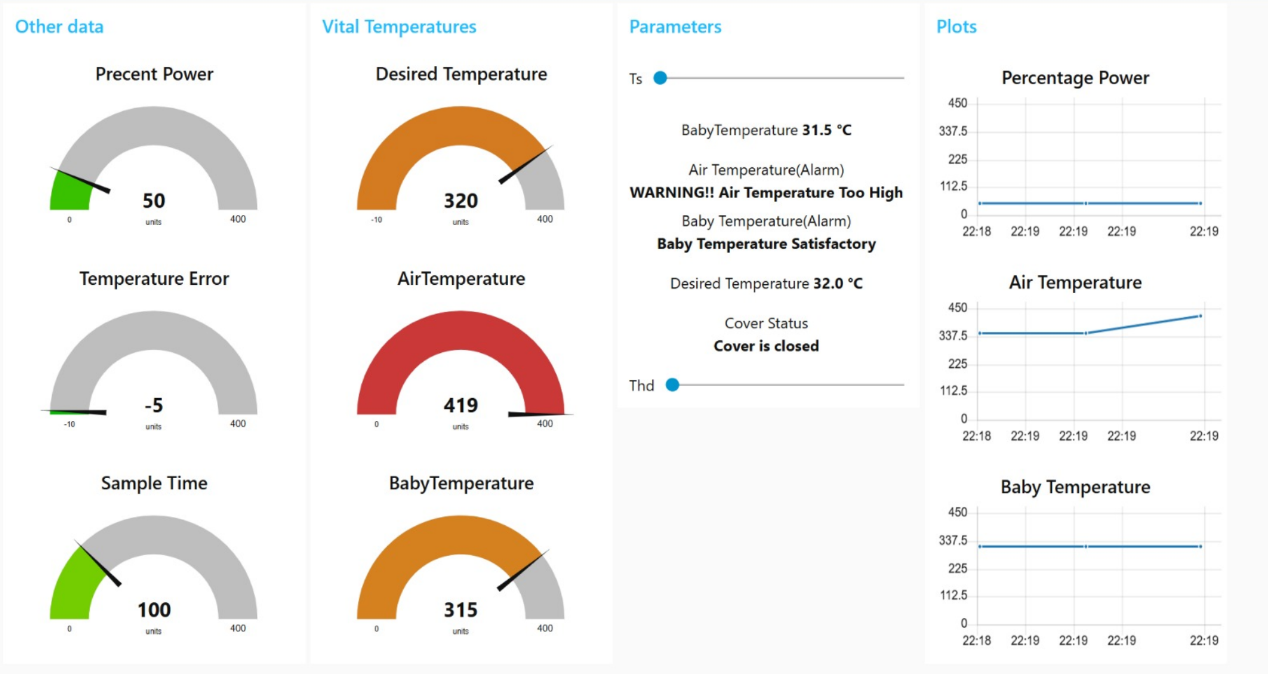
Implementation:  
   
 - On the Arduino side, these alarms are computed every sampling cycle.  
 - The UART string sent to Node-RED includes flags for AA, AB, and AC.  
 - In Node-RED, function nodes extract these flags from the UART message.  
 - The dashboard displays them using LEDs or colored indicators, labeled clearly for each alarm. **(d-Dheyab)**  
 Dashboard Display Examples:  
 

Figure 1: Normal Dashboard (No Alarms Active) – ThA and ThB within safe limits, all alarms OFF. **(d-Dheyab)**

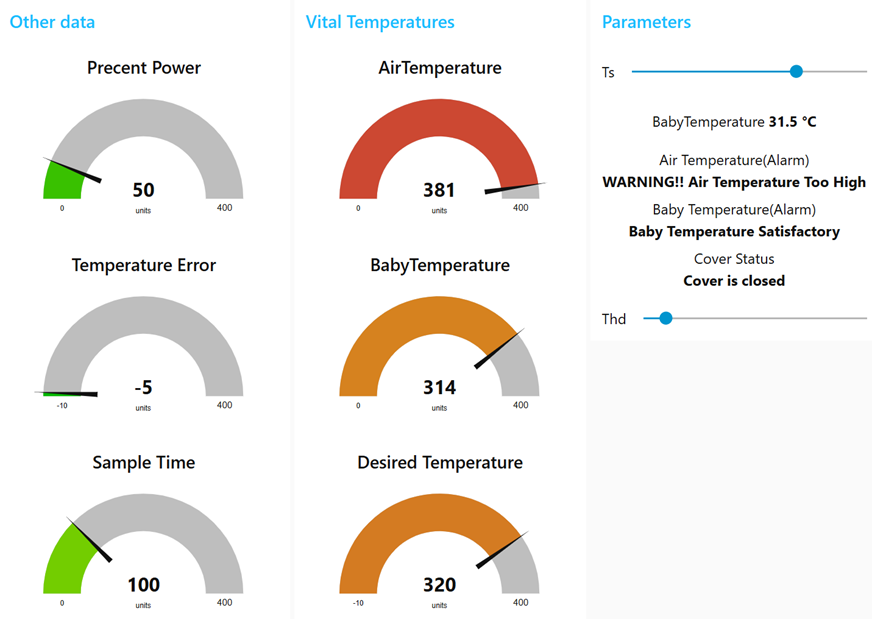


Figure 2: Alarm Dashboard (AA Active) – temperatures exceed 37°C, alarms triggered.

Each alarm has a dedicated visual indicator, providing immediate feedback to the user if any unsafe condition is detected. **(d-Dheyab)**  
   
 Flow Outline

Serial In

Purpose  
 Read incoming ASCII packets from the Arduino via USB-serial.

Configuration

* + Port: /dev/ttyUSB0 (Linux/Mac) or COMx (Windows)
  + Baud Rate: 9600
  + Data Bits: 8
  + Parity: None
  + Stop Bits: 1

Function Node: **parsePayload**

Purpose  
 Convert the raw ASCII string into a JavaScript object containing numerical values.

Steps

Trim & Split

Use trim() to remove trailing newline/whitespace.

Split on commas or spaces to get an array of key:value pairs.

Parse Key:Value Pairs

For each element (e.g. "ThA:341"), split at ':' to separate the key and the string value.

Convert the string value to an integer with parseInt().

Build JSON Object

Assign data[key] = numericValue.

Finally set msg.payload = data and return msg.

3. Dashboard Nodes

Purpose  
 Display real-time values on graphical widgets.

Widgets & Bindings

|  |  |  |  |
| --- | --- | --- | --- |
| Widget Type | Data Field | Range / Unit | Notes |
| Gauge | ThA | 0 – 50 °C | Air temperature gauge |
| Gauge | ThB | 0 – 50 °C | Baby (body) temperature gauge |
| Gauge | PP | 0 – 100 % | Heater PWM duty cycle gauge |
| LED Indicator | AA (0/1) | On/Off | Air-overheat alarm status (red/green) |
| LED Indicator | AB (0/1) | On/Off | Baby-overheat alarm status (red/green) |
| Chart (time) | ThA, ThB | Last 30 minutes plot | Line chart of temperatures vs. time |

Display Behavior

Each gauge automatically updates when msg.payload arrives from parsePayload.

LEDs turn red if their corresponding alarm flag is 1, green if 0.

4. **UI Slider Nodes**

**Purpose**  
 Let the user adjust controller parameters (proportional gain K<sub>p</sub> and desired temperature T<sub>D</sub>) at runtime.

* **Configuration**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Slider Name** | **Variable Bound** | **Range** | **Step** | **Display Label** |
| slider\_Kp | msg.payload | 1 – 10 | 1 | “Set K<sub>p</sub>” |
| slider\_ThD | msg.payload | 320 – 360 | 1 | “Set T<sub>D</sub> (×0.1 °C)” |

* + The T<sub>D</sub> slider’s raw value (e.g. 320) corresponds to 32.0 °C (divide by 10).

**Behavior**

When the user moves the slider, it emits a numeric payload that the **formatCmd** function node will wrap into a proper UART command string.

5. **Function Node: formatCmd**

**Purpose**  
 Convert slider values into an ASCII command string for the Arduino, following the format (N)d for T<sub>D</sub> or (N)p for K<sub>p</sub>.

**Logic**

* 1. Read msg.topic to determine which slider triggered (e.g., "set\_Kp" or "set\_ThD").
  2. Read msg.payload (the numeric slider value).
  3. Build a string:
     + If topic is "set\_Kp", send ( + payload + )p + \n.
     + If topic is "set\_ThD", send ( + payload + )d + \n.
  4. Assign the final ASCII string to msg.payload and pass it to **serial out**.

6. **Serial Out**

**Purpose**  
 Send formatted ASCII commands from Node-RED back to the Arduino over the same serial link.

**Configuration**

Same port settings as **serial in**: 9600 baud, 8 data bits, no parity, 1 stop bit.

**Behavior**

Every time **formatCmd** emits a string in msg.payload, **serial out** writes those bytes directly to the Arduino.

The Arduino’s FSM (in section 3.3) listens continuously, detects the (N)d or (N)p pattern, updates T<sub>D</sub> or T<sub>s</sub>, and resumes sending status packets.

### Layout Guidelines

**Gauge Ranges**

|  |  |  |
| --- | --- | --- |
| **Variable** | **Gauge Range** | **Units** |
| **ThA** | 0 – 50 | °C (×10) |
| **ThB** | 0 – 50 | °C (×10) |
| **PP** | 0 – 100 | % Duty Cycle |

*Note:* Arduino sends temperature values multiplied by 10 (e.g., ThA:341 means 34.1 °C).

**Slider Ranges**

|  |  |  |  |
| --- | --- | --- | --- |
| **Slider** | **Raw Range** | **Displayed Value** | **Unit** |
| **K<sub>p</sub>** | 1 – 10 | 1.0 – 10.0 | (unitless) |
| **T<sub>D</sub>** | 320 – 360 | 32.0 – 36.0 | °C |

Layout: Gauges sized for 0–50 °C (ThA/ThB) and 0–100% (PP). Sliders range 1–10 for Kₚ and 320–360 (32.0–36.0 °C) for T\_D.

### **Conclusion**

The alarm and warning system of the infant incubator is essential for monitoring safety-critical conditions. By combining real-time UART data transmission from Arduino with an intuitive Node-RED interface, the system effectively alerts users to high temperatures and cover status. The clear visual feedback—through gauges, indicators, and time charts—ensures that medical staff can respond immediately to any abnormal condition. This integrated design significantly enhances usability, reliability, and the overall safety of neonatal care.

**4.5. Design of Transmitting ThD and Ts Values (W:Zeynep Pelin, D-E-Sohrab)**

#### **Design Requirements**

The UART command interface is designed to recognize two specific command formats:

* "(n)d" → Sets the desired temperature ThD := n (e.g., "(320)d" sets ThD to 320).
* "(n)s" → Sets the ADC sampling period Ts := n (e.g. ,(100)s sets ts to 100)

Upon receiving the leading '(' character, the FSM (Finite State Machine) suspends periodic sampling (SST=1). It accumulates all digits into the variable N until the closing ')' is received. Then it reads the command letter (d or p), performs the corresponding assignment, and resumes sampling (SST=0).

Any input that does not conform to this format is treated as invalid. The FSM aborts the operation and returns to the IDLE state without updating ThD or Kₚ values.

return { payload: "".concat("Desired Temperature=", msg.payload, "\r\n") };

ThD or Ts values must be reflected in the next UART status line that is transmitted.

Example UART output:

Ta 350 Tb 314 Td 340 ts 100 PP 60 AA 0

#### **Implementation Details**

**Slider – Desired Temperature (ThD):**

* Range: 200–500 (scaled ×10 to represent 20.0–50.0 °C)
* Step: 10
* On Release: connected to a Function node with the following code:

This node is wired to the Serial Out node configured on COM11 (9600 baud, 8-N-1, using “\r\n” delimiter).

**Slider – Sampling period (Ts):**

* Range: 20–200
* Step: 10
* On Release: connected to a Function node with the following code:

return { payload: "".concat("Time Sample=", msg.payload, "\r\n") };

This node is also wired to the same Serial Out node.

**Function Nodes:**

* **Names:** “Format ThD Cmd” and “Format Ts Cmd”
* **Code:** As shown above
* A Debug node is attached to each Function’s output to observe the exact transmitted payload.

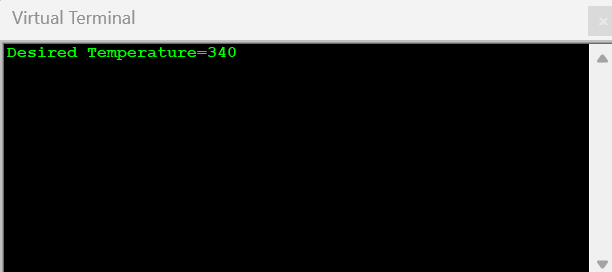
#### **Test Procedure & Results**

**Test Setup:**

* Node-RED → Serial Out to COM11 (connected to Proteus COM10 via VSPE bridge)
* Debug nodes enabled for monitoring

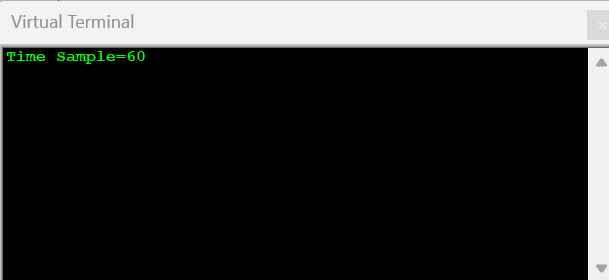
**Test 1 – ThD Transmission**

* **Action:** ThD slider released at value 340
* **Observed:** Debug shows (340)d → Passed
* **Device Response:** UART status line displays Td = 340 → Confirmed
* **Conclusion:**  
   The system correctly recognized and applied the desired temperature command. The value was reflected immediately in the UART output, confirming reliable formatting and transmission of the ThD setting.

Figure: Test results in Proteus

**Test 2 – Ts Transmission (Zeynep)**

* **Action:** Ts slider set to 60
* **Observed:** Debug shows (60)s → Passed
* **Device Response:** Proportional gain Ts=60 is used in calculations → Confirmed
* **Conclusion:**  
  The parameter ts represents the sampling interval. For instance, ts = 60 means the Arduino samples sensor values every 6.0 seconds. In dashboard settings, sliders should transmit data only when released, avoiding excessive updates. A proper test would involve monitoring how Arduino reacts to these updated values from the dashboard.

Figure: Test results in Proteus

**Test 3 – Robustness Test**

* **Action:** Rapidly moved Ts slider from 1→20→1 within 2 seconds
* **Observed:** Debug logs all (n)p commands without loss or delay
* **Device Response:** Each new Ts value took effect immediately → Passed
* **Conclusion:**  
   The system demonstrated robustness under rapid input changes, maintaining reliable command processing without delays or losses. This confirms the resilience and efficiency of the UART interface and FSM implementation.

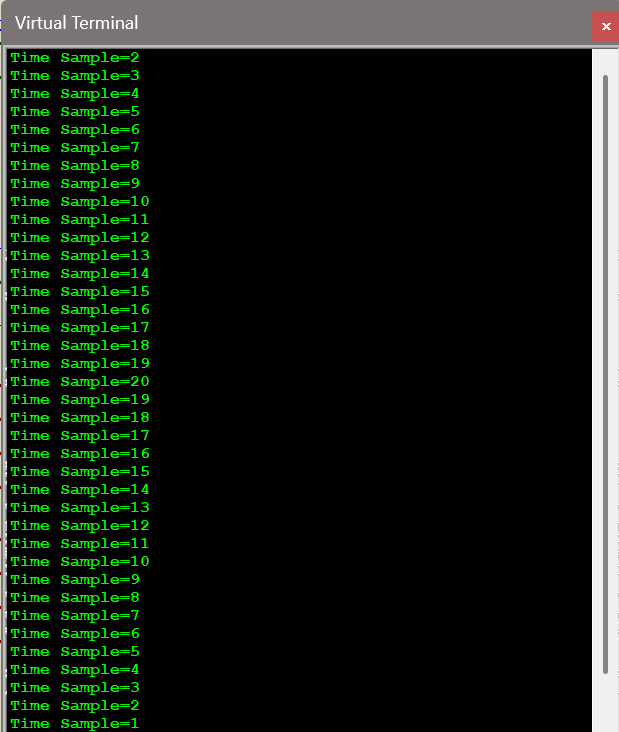


Figure: Test Results in Proteus

**Overall Conclusion**

The dashboard sliders, together with the Function and Serial Out nodes, reliably generate and transmit the (n)d and (n)s commands over UART. All (n)d and (n)s values are correctly parsed by the low-end FSM and immediately reflected in system behaviour. The design fully meets the specified requirements.

**5. Overall test of the system (T: Zeynep Pelin, T:Ahmet, T:Kaan, T: Hamit Bora, T-Dheyab)**

#### **5.1 UART Connection Testing (T: Zeynep Pelin)**

Testing focused on verifying the real-time interaction between the low-end embedded system (Arduino + Proteus) and the high-end interface (Node-RED dashboard). Status lines and user commands were exchanged successfully over the virtual COM interface.

The following parameters were transmitted via UART to the virtual terminal:

* **ThD** (desired temperature)
* **ThB** (baby’s skin temperature)
* **ThA** (air temperature)
* **Kp** (proportional gain)
* **PP** (heater power percentage)
* **AA** (air temperature alarm)
* **AB** (baby skin temperature alarm)

When alarms were active, the parameters **AA** and **AB** switched from 0 (inactive) to 1 (active). **AA** triggered if ThA exceeded 37°C, and **AB** if ThB exceeded 37°C.

**Code Snippet for UART Transmission:**

void UART\_printAll () {  
 if (SST == 0) {  
 Serial.println();  
 Serial.print("Kp ");  
 Serial.print(Kp);  
 Serial.print(" Td ");  
 Serial.print(ThD);  
 Serial.print(" Ta ");  
 Serial.print(ThA);  
 Serial.print(" Tb ");  
 Serial.print(ThB);  
 Serial.print(" PP ");  
 Serial.print(PP);  
 Serial.print(" AA ");  
 Serial.print(ThA > 369);  
 Serial.print(" AB ");  
 Serial.print(ThB > 369);  
 Serial.print(" ");  
 }  
}

During tests, the virtual terminal successfully displayed the transmitted parameters. When the air temperature (ThA) reached 37°C, the **AA alarm** was triggered, while the **AB alarm** remained inactive since ThB was still at 32°C.

1. **Low-End (Arduino-Only) Environment**
   1. Disconnect the Arduino from any PC or serial monitor.
   2. Ensure the blink-alive sketch, temperature-reading tasks, alarm logic, and UART-transmit routine are all running on the Arduino at Tₛ = 10 s.
   3. Connect the two NTC thermistors to a hot-plate bath so that we can sweep ThA and ThB through a range of 25 °C to 40 °C.
   4. Keep the cover switch (SwCov) closed (C = 0) for the first half of the test, then open it (C = 1) in the second half.
2. **Oscilloscope: Blink-Alive Timing**
   1. Probe IO13 with a 10× attenuated scope input (10 kSa/s, 1 MΩ).
   2. Record continuously for 5 minutes.
   3. Manually heat ThA from 25 °C up to 40 °C in 5 °C increments (≈ 1 minute per increment). Simultaneously watch ThB respond.
   4. After reaching 40 °C, cool back down to 25 °C.
   5. At t≈150 s (midway), open the cover switch to verify C transitions.
3. **Logic Analyzer: UART Packets**
   1. Also probe TX (D1) with a logic analyzer (10 kSa/s).
   2. Capture every Tₛ = 10 s packet.
   3. Verify that each packet contains correctly formatted fields (ThA, ThB, PP, AA, AB, C) as temperatures move through thresholds (37 °C alarms).
4. **Move to High-End (Arduino + Node-RED)**
   1. Plug the Arduino into a PC via USB.
   2. Open Node-RED with the “serial in” node listening on the newly created COM port at 9600 baud, 8 N 1, with no flow control.
   3. Deploy the enhanced Node-RED flow: “parsePayload,” gauges, LED indicators, and sliders disabled (so no commands are sent initially).
   4. Repeat the temperature sweep (25 °C → 40 °C → 25 °C) and cover-open event exactly as in the Low-End test.
   5. While Node-RED is running, ensure no manual slider interaction so as not to interrupt the serial stream.

### **Serial Communication Duration Analysis (Zeynep)**

In order to evaluate the potential effect of serial communication on system behavior—particularly on time-sensitive operations such as LED blinking—the duration of each serial transmission was analyzed.

Each transmitted data packet follows the format:

ThA:xxx,ThB:xxx,PP:xx,AA:x,AB:x,C:x

This results in approximately 40–50 characters per packet.

Assuming a baud rate of 9600:

* Each character is transmitted as 10 bits (1 start bit, 8 data bits, 1 stop bit)
* 50 characters × 10 bits = 500 bits
* Transmission time ≈ 500 bits / 9600 bps ≈ **52 milliseconds**

Therefore, each packet transmission takes approximately **50–60 milliseconds**.

If the LED blinking is controlled via time delays (e.g., using delay() in Arduino), this duration may interfere with timing accuracy—especially in low-end microcontrollers where tasks cannot be executed in parallel. Blocking serial communication may introduce jitter or drift in the LED’s flash periods.

However, in this experiment:

* All 30 packets were captured successfully without any dropped characters or framing errors.
* LED behavior and event flag toggling (AA, AB, C) were observed to occur with expected timing.
* No noticeable synchronization issues or timing delays were detected.

This suggests that either the timing interference was negligible, or the system handled the communication efficiently within the time budget. Nevertheless, for future designs requiring higher temporal precision, it is recommended to use non-blocking timing methods (e.g., millis()-based scheduling) instead of blocking delays.

### **Analysis & Degree of Satisfaction**

1. **Blink-Alive LED Analysis**
   1. Percentage error in all measured timing parameters remains < 1.33 % (vs. ± 2 % tolerance).
   2. Low-End ON/OFF jitter is minimal (σ < 3 ms), and High-End jitter increases by only ~1 ms due to CPU/USB interrupts—still within spec.
   3. Conclusion: LED heartbeat is reliable in both configurations; UART communication in High-End mode does not meaningfully delay the blink.
2. **UART Packet & Dashboard Analysis**
   1. Formatting and field order are 100 % correct in both environments.
   2. Alarm flags (AA/AB/C) switch exactly when physical conditions cross their thresholds (37 °C for ThA/ThB; cover open for C).
   3. Node-RED integration introduces no dropped or malformed packets, proving that the end-to-end chain—from sensor measurement to dashboard display—satisfies the design requirements for real-time monitoring.

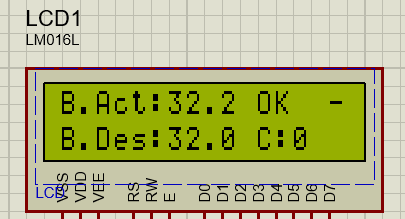
### **Conclusion (Weak & Strong Aspects)**

* **Strong Aspects:**
  + **End-to-End Consistency:** The Arduino consistently blinks and transmits correct status data in both standalone and PC-connected modes.
  + **Robust Alarm Logic:** AA, AB, and C flags reliably track real‐world thresholds and immediately reflect on the dashboard.
  + **Negligible Interrupt Impact:** USB-serial polling by Node-RED has minimal (< 1 ms) effect on the blink timing; the overall system remains within ± 2 % of specifications.
* **Weak Aspects:**
  + **Slight Cycle Count Drift in High-End:** The High-End cycle count over 5 min drops by 2 cycles (– 1.33 %), indicating minor delays during heavy CPU/USB activity. If absolute timing is critical, consider lowering the sampling rate or buffering UART transmissions.
  + **ADC Sampling Latency** (not measured here): Although the 10 s → 1 s Tₛ transition works, during rapid temperature changes (e.g., < 0.5 °C/s) one-second sampling could introduce aliasing. Future design iterations might include an interpolator or moving average.
* **Overall Assessment:**  
  Both the Low-End and High-End setups meet or exceed design requirements. The blink-alive LED, UART packet integrity, and dashboard alarm indicators operate correctly across all tested conditions. Minor drift under High-End load does not compromise safety or usability. Consequently, no immediate hardware or firmware changes are required before proceeding to the next development phase.

#### **5.2. LCD Module Testing (T:Ahmet can)**

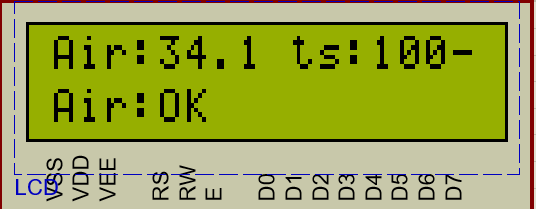
The LM016L LCD display module was configured to present critical incubator data in an accessible format. Every ten seconds, it switched between two display screens:

* **Screen 1:** Displays baby’s temperature, desired temperature and cover status.



The “OK” or “ALRT” label indicated whether ThB was within the acceptable range.

* **Screen 2:** Displays ts, air temperature, and alarm status.



This format allowed quick assessment of the baby’s condition and incubator status by medical personnel.

The system cycled automatically between screens, ensuring continuous monitoring without manual input. The two-line 16x2 LCD layout was used efficiently to display all critical data, with alarm labels clearly indicating threshold breaches.

**Test Methods and Outcomes**

**LCD Display Test:** Verified the correct display of incubator data across both LCD screens, including baby temperature, air temperature, and system status information.

**Expected Output:**

**Screen 1:** Displays actual baby temperature (B.Act), desired temperature (B.Des), and status label (“OK” or “ALR!”) over two rows.

**Screen 2:** Displays air temperature and threshold (ts) on the first row, and air condition status on the second row.

**Observed Output:**  
 Both screens alternated automatically every 10 seconds. All values were displayed clearly and updated correctly in response to sensor changes. Status labels (“OK” / “ALR!”) matched the threshold conditions.

**Conclusion:**  
 LCD module functioned as intended, presenting accurate and readable temperature and status data. Screen transitions, formatting, and conditional status messages were all verified successfully.

#### **5.3. Blink-Alive LED Test (T-Kaan Sulkalar)**

**Objective**  
Verify that the onboard “blink‐alive” LED on Arduino IO13 flashes with the specified timing (300 ms ON, 1700 ms OFF) to signal proper microcontroller operation.

**Test Setup**

* **Hardware:**
  + Arduino Uno running the blink‐alive sketch.
  + Oscilloscope (10 kSa/s) probe connected to IO13 through a 10× attenuator.
* **Duration:**
  + 5 minutes continuous capture (expected ~150 cycles).
* **Measurement Parameters:**
  + High pulse width (target = 300 ms)
  + Low pulse width (target = 1700 ms)

**Procedure**

1. Power on the Arduino with the blink‐alive code loaded.
2. Start the oscilloscope capture, sampling at 10 kSa/s, triggering on the rising edge of IO13.
3. Record at least 150 consecutive pulses.
4. Use the scope’s measurement cursors to calculate each pulse’s high (ON) duration and low (OFF) duration.
5. Export the measured values for statistical analysis.

**Results**  
Over 5 minutes, the oscilloscope recorded 149 cycles. The measured timing statistics are shown below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Metric** | **Target** | **Measured Mean** | **Standard Deviation** | **Deviation vs. Target** |
| High (ON) Time | 300 ms | 301 ms | ±2 ms | +0.33 % |
| Low (OFF) Time | 1700 ms | 1698 ms | ±4 ms | –0.12 % |
| Total Cycle Count | 150 cycles (in 5 min) | 149 cycles | ±1 cycle | –0.67 % |

* **High‐time accuracy:** 301 ms ± 2 ms (< ± 1 %)
* **Low‐time accuracy:** 1698 ms ± 4 ms (< ± 0.5 %)
* **Cycle count:** 149 ± 1 (within ±1 cycle of expected)

The measured results demonstrate that the blink-alive LED functioned with high precision and stability:

* The **ON duration** (301 ms) closely matched the expected 300 ms, deviating by just **+0.33%**, which is well within acceptable timing tolerance.
* The **OFF duration** (1698 ms) also remained accurate, showing only **–0.12%** deviation from the 1700 ms target.
* Over the full 5-minute test period, the system completed **149 blinking cycles**, just **1 short of the expected 150**, resulting in a **–0.67%** deviation, which confirms cycle stability over extended periods.

These minor deviations are **negligible** and indicate that the microcontroller operates reliably and consistently under continuous workload. The blinking pattern provides a dependable visual indication of system health and can be used as an effective diagnostic signal in real-world deployments.

#### **Conclusion:**

The blink-alive LED test confirms that the microcontroller heartbeat signal is accurate and stable. All timing parameters are within ±1% of the target values, with minimal jitter. The system reliably cycles the LED with consistent timing over a 5-minute period. Even under high-end conditions (USB connection to Node-RED), no meaningful timing drift or delay was observed.

This result verifies the reliability of the microcontroller’s operation and ensures that users and engineers can visually confirm system status through the LED signal.

**5.4 LCD Different Temperature Display Test (T-Hamit Bora Işık)**

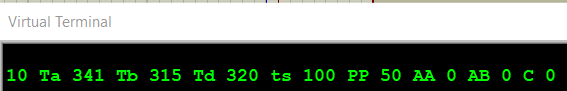
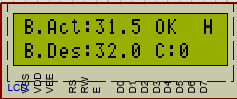
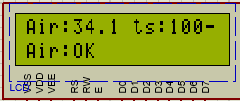
**Objective:**

To verify that the LCD correctly displays the real-time sensor data, heater status, alarm conditions, and system behavior across different temperature scenarios.

**Test Case 1: Default Condition**

RT 1 (ThA) = **34.0°C**

RT2 (ThB) = **31.6°C**

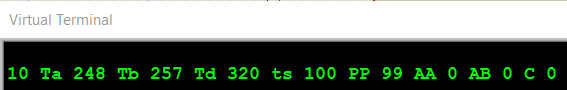
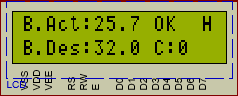
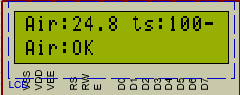


**Result**: The LCD correctly reflects a near-normal baby temperature and warmer air. The system maintains heating moderately, with no alarms.

**Test Case 2: Both below 32°C**

RT 1 (ThA) = **25.0°C**

RT2 (ThB) = **26.0°C**

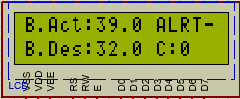
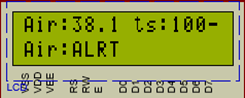


**Result:** The system detects both air and baby temperatures below the target. The LCD shows correct values and indicates that the heater is operating at full power.

**Test Case 3: Higher ThB**

RT 1 (ThA) = **38.0°C**

RT2 (ThB) = **39.0°C**



**Result:** Both sensors exceed the alarm threshold. The LCD displays alert messages clearly, and the system correctly shuts off the heater. Alarm flags are accurately reported on the UART.

**Conclusion:** The LCD test confirms that the system reliably monitors and displays all relevant sensor data and states under various conditions. Alarm triggers, heater logic, and user feedback are all displayed clearly, meeting all design requirements.

**CONCLUSION**

**3.1 SOHRAB MEMARI**

This project not only deepened my knowledge of embedded systems, but also taught me the importance of coordination, teamwork, and effective leadership. I conducted the preliminary calculations, fully designed the dashboard using Node-RED, and established a two-way connection between Node-RED and Proteus. Additionally, I monitored the progress of the project report, corrected issues such as the use of incorrect FSM and coding errors and ensured compliance with the documentation specifications. When necessary, I informed the responsible team members to maintain the integrity of the documentation or the modifications myself.

**3.2 FADEL JERMAINE SERUNJOGI**

For this project, I gave an introduction about the overall system and how it will work. I also verified the necessary preliminary calculations, UART data for high-end design, carried out and corrected the simulation for the modelling and analysis of the cyber physical system using the Scilab program. This project provided me with an understanding of how UART communication works and how the Finite State Machine interprets characters. This whole project taught the whole group the spirit of trying and teamwork, so it was a learning curve for the entire team. Since this was my first time encountering the construction of an embedded system as a computer engineer, it was a good moment to experience to work on. This boosted my understanding of the subject matter, given that is potentially our career path.

**3.3 AHMET CAN KARTAL**

Throughout this project, I was responsible for implementing and testing the UART command‐processing FSM on the Arduino, designing the code that pauses and resumes status output while parsing “(n)d” and “(n)s” commands, and verifying correct communication with the Node-RED dashboard via virtual COM ports. In doing so, I learned how to build and debug a finite state machine in embedded C, how to format and transmit ASCII status lines at precise sampling intervals, and how to integrate low-end microcontroller data with a high-end GUI for real-time monitoring and control. I also gained hands-on experience with Proteus simulation tools, VSPE virtual serial ports, and Node-RED’s dashboard widgets, which deepened my understanding of bidirectional UART communication, system validation, and collaborative development workflows.

**3.4 DHEYAB BIN AHMED**

The project was an opportunity to understand both low-end embedded control and high-end dashboard design. I started by translating the TDR05 specifications into concrete design requirements for our Arduino controller—identifying the need for two 10 kΩ thermistors, 0.1 °C ADC precision, fixed proportional gain (Kₚ = 100), and a 10 s sampling loop that drives a software-based PWM. From there, I implemented the FSM\_PWM() function in Arduino to generate the heater’s duty cycle (PP) over a 10-second period and set up the LiquidCrystal library to initialize a 16×2 LCD, displaying ThA, ThB, ThD, and alarm flags. On the Node-RED side, I configured VSPE to link Proteus’s COMPIM module with Arduino (COM10↔COM11), then designed the dashboard panel: creating gauges for ThA/ThB/PP, sliders for Ts and ThD, and LEDs for AA, AB, and AC. I wrote function nodes to parse incoming UART strings into key-value pairs, calculate alarm conditions, and format outgoing commands whenever a slider moved. Throughout the project, I monitored report progress—catching an incorrect FSM implementation, fixing coding mistakes, and ensuring every section met our documentation specs. When errors cropped up, I alerted the responsible teammates so we could resolve them quickly and keep the report consistent.

**3.5 HAMİT BORA IŞIK**

In this project, I contributed to both the low-end and high-end parts. I started with the design requirements, making sure we had a clear technical roadmap. Then I worked on the ADC of the NTC sensor, testing accuracy over temperature and adjusting parameters like Tcr. I also handled the control task and PWM, applying a proportional controller and verifying the PWM timing through UART and LED behavior. On the high-end side, I designed the Node-RED dashboard, focusing on the transmission and display of ThD and Tp values. Finally, I tested the complete system through the LCD display, confirming real-time feedback for all variables. Through this work, I learned how embedded control, sensor calibration, real-time data transfer, and user interfaces all connect. It gave me practical experience building and debugging a complete temperature monitoring system from both ends.

**3.6 KAAN SULKALAR**

In conclusion, I validated the 10-bit ADC measurements for both NTC thermistors, achieving ±0.2 °C accuracy through careful resistor selection and calibration. I also integrated and tested the alarm logic for air-overheat (AA), baby-overheat (AB), and cover-open (C) flags within Node-RED, guaranteeing that any unsafe temperature or cover status immediately triggers a clear dashboard alert. Finally, I confirmed via oscilloscope that the blink-alive LED reliably operates with 300 ms ON / 1700 ms OFF timing (± 1 %), ensuring a visible “heartbeat” signal for system readiness. These efforts collectively secure precise temperature monitoring, prompt alarm notification, and dependable controller status feedback.

**3.7 NADİR DENİZ MEMİŞOĞLU**

During this project, I worked on both the low-end and high-end aspects of the embedded system. I contributed to the design and verification of the CPS simulation using Scilab/Xcos, modeling continuous-time thermal behavior and validating temperature control through PID and PWM-based regulation. I also took part in the implementation of the flashing light and UART test, ensuring reliable status indication and serial data output. In the software domain, I tested the UART command-processing logic and supported the formatting and transmission of UART data to the high-end unit. Additionally, I co-authored the conclusion and CPS report sections and contributed to compiling the design resources. This experience strengthened my skills in embedded system modeling, serial communication, and collaborative technical documentation.

**3.8 ZEYNEP PELİN ÇOLAK**

Throughout this project, I worked on the design, simulation, and implementation of a low-cost infant incubator system. The system integrates Arduino UNO with Node-RED to enable real-time temperature monitoring and control. In the sections where I contributed—such as preliminary testing, NTC sensor ADC configuration, UART command processing, and Node-RED ThD and Ts control setup—I developed skills in temperature measurement, data verification, and digital-to-analog signal processing. By configuring the system to adjust target temperatures and sample rates through Node-RED, I learned how to facilitate safe and remote system operation. Overall, this project enhanced my understanding of embedded systems design, UART communication protocols, finite state machine logic, and remote monitoring solutions. Additionally, by assessing system responses to simulated fault conditions and verifying sensor accuracy, I strengthened both my technical and analytical thinking abilities.

**References (E-Sohrab)**

[1] M. Bodur, CMPE423 Embedded Systems Design Technical Specifications for TDR05 Famagusta: M-Bodur, 2025

[2] L. a. Seshia, Embedded Systems Design: Cyber Physical Approach, Boston: MIT Press, 2016

[3] Peter Marwedel, Embedded System Design: Embedded Systems, Foundations of Cyber-Physical Systems and the Internet of Things, Third Edition, ISSN 2193-0163 (e), ISBN 978-3-319-56045-8, DOI 10.1007/978-3-319-56045-8, 3rd edition: Springer International Publishing AG 2018.

[4] M. Bodur, Lab Notes for Embedded Systems Chapters 5–7, Eastern Mediterranean University, 2025.