A blue and white logo

AI-generated content may be incorrect.

Ministry of Education and Research of the Republic of Moldova

Technical University of Moldova

Department of Software and Automation Engineering

**REPORT**

Laboratory work No. 3.2

**Discipline**: Embedded Systems

Elaborated: Berco Andrei, FAF - 221

Checked: asist. univ., Martiniuc A.

Chișinău 2025

**Analysis of the Situation in the Field**

1. **Description of the Technologies Used and Application Context**

Embedded systems are pivotal in real-time data acquisition and processing, enabling applications like environmental monitoring, industrial automation, and IoT. This lab work demonstrates:

* **Signal conditioning** for analog/digital sensors (e.g., noise filtering, scaling).
* **FreeRTOS** for deterministic task scheduling (e.g., vTaskDelayUntil for precise sampling).
* **STDIO** (via printf) for reporting data to LCD/Serial.

**Application Context**:

* **Industrial IoT**: Monitor temperature/humidity with filtered, real-time data.
* **Smart Agriculture**: Process soil moisture sensor data with adaptive filtering.

**Overview of the Hardware and Software Components Used**

A screenshot of a computer program

AI-generated content may be incorrect.

**System Architecture Explanation and Solution Justification**

**Task Breakdown**

1. **Task\_SensorRead**
   * Reads ADC (analog) or I2C (digital) at fixed intervals using vTaskDelayUntil().
   * Example: adc\_val = analogRead(SENSOR\_PIN);
2. **Task\_FilterData**
   * **Salt-and-Pepper Filter**: Removes outliers (e.g., if (value > threshold) discard).
   * **Weighted Moving Average**: Smooths data (e.g., filtered = (0.6\*current + 0.4\*previous)).
3. **Task\_Display**
   * Converts ADC to voltage: Vout = (adc\_val / 4095) \* 3.3V.
   * Scales to physical units (e.g., °C for LM35: Temp = Vout \* 100).
   * Outputs via printf("Temp: %.2f°C\n", temp); every 500ms.

**Why FreeRTOS?**

* **Precision**: Fixed sampling intervals avoid jitter.
* **Modularity**: Isolate sensor read, processing, and display tasks.
* **Scalability**: Add tasks (e.g., wireless transmission) without disrupting timing.

1. **Case Study: Sequential Task Execution in Embedded Systems**

**Context and Necessity**

Many embedded applications (e.g., weather stations, smart agriculture) require **periodic sensor readings** with minimal latency. This case study demonstrates a **FreeRTOS-based** approach for reliable signal acquisition

**Practical Implementation**

1. **Task 1: Sensor Read**
   * Analog sensors: analogRead() → scaling (e.g., Vout = (ADC\_val / 4095) \* 3.3V).
   * Digital sensors: I2C/SPI communication (e.g., BME280.readTemperature()).
   * Uses vTaskDelayUntil() for **consistent sampling intervals**.
2. **Task 2: Data Processing**
   * Applies calibration/formulas (e.g., Temp = (Vo \* 100) / 1024).
   * Stores results in a **global struct** for other tasks.
3. **Task 3: Display & Reporting**
   * Prints formatted data via printf() every **500ms**:

**Extending the Case Study**

* **Wireless Transmission** – Integrate Wi-Fi (ESP32) to send data to a cloud dashboard.
* **Multi-Sensor Fusion** – Add more sensors (CO₂, pressure) with independent tasks.
* **Energy Optimization** – Use vTaskSuspend() in battery-powered scenarios.

**Benefits and Impact**

* **Structured Timing** – FreeRTOS ensures no task starvation.
* **Scalability** – Easy to add more sensors/tasks.
* **Debugging-Friendly** – STDIO provides real-time logs.**Scalability** – Can be integrated with wireless communication, IoT frameworks, or real-time monitoring systems.

This case study highlights the importance of structured scheduling in embedded applications, demonstrating how sequential execution models enhance reliability and efficiency in task management.

**Design**

1. **Architectural Sketch and Component Interconnection**

The system consists of:

* **MCU (ESP32/Arduino)** – Core processing unit.
* **Sensor** – Connected via:
* Analog: Direct to ADC pin.
* Digital: I2C (SCL/SDA) or SPI (MISO/MOSI/SCK).
* **Serial Monitor** – For debugging (printf output).

A screenshot of a computer screen

AI-generated content may be incorrect.

*Figure 2.1 Component diagram*

**A circuit board with a cable connected to it

AI-generated content may be incorrect.**

*Figure 2.2 Component scheme*

A computer screen shot of a circuit board

AI-generated content may be incorrect.

*Figure 2.3 Electrical scheme*

1. **Scheme bloc and algorithm**

To understand the system's behavior, a Flowchart and a Finite State Machine (FSM) are used.

**Flowchart – Serial Command Processing**  
(A Flowchart diagram illustrating the cycle: command reception → processing → execution → user feedback)

A screenshot of a computer

AI-generated content may be incorrect.

*Figure 2.4 Flowchart*

**Explanation:**

1. **Initialization: Sets up hardware (Serial, Ultrasonic sensor) and FreeRTOS (tasks, mutex).**
2. **Ultrasonic Task (Cyclic):**
   * **Measures distance → Filters (salt & pepper → weighted average) → Saturates → Prints.**
   * **Uses vTaskDelayUntil() for precise timing.**

A screenshot of a diagram

AI-generated content may be incorrect.

*Figure 2.4 FSM diagram*

* **Initialization**: Configures pins and buffers.
* **Idle**:
  + WaitTrigger: Waits to send ultrasonic pulse.
  + MeasureDistance: Captures echo duration.
* **FilterData**:
  + SaltPepperFilter: Removes outliers via median.
  + WeightedAverage: Smooths with weighted window.
* **Saturation**: Ensures value is within [MIN\_DISTANCE\_CM, MAX\_DISTANCE\_CM].
* **Output**: Prints via printf (protected by mutex).

1. **Modular implementation**

For better project organization, a modular architecture was used, dividing functionalities into separate files.

A screenshot of a computer program

AI-generated content may be incorrect.

*Figure 3.1 Project organization*

The project is well-structured, separating functionality into different modules using a layered approach. The organization follows best practices for embedded systems and modular programming.

#### **Code Functionality Overview:**

1. **main.cpp**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Figure 3.1: Flowchart for main.cpp**

**Description**

* **Purpose**: Coordinates tasks using FreeRTOS.
* **Key Sections**:
  1. **Setup**:
     + Initializes Serial, sensor, and filter buffers.
     + Creates a mutex to protect printf.
  2. **FreeRTOS Task** (ultrasonic\_distance\_task):
     + **Cyclic Workflow**:
       1. Read sensor → Filter → Saturate → Print.
       2. Uses vTaskDelayUntil() for precise 100ms intervals.
  3. **Filter Pipeline**:
     + Raw → Salt & Pepper → Weighted Average → Saturation.
* **Critical Components**:
  1. data\_mutex: Prevents Serial corruption during printf.
  2. STACK\_SIZE: Reduced to 128 bytes (Arduino Uno memory limits).

1. **filter.cpp**

A diagram of a flowchart

AI-generated content may be incorrect.

*Figure 3.2: Flowchart for**filter.cpp*

**Description**

* **Purpose: Implements digital filters for signal conditioning.**
* **Functions:**
  1. **apply\_salt\_pepper\_filter():**
     + **Shifts a sliding window buffer.**
     + **Sorts values (bubble sort) and returns the median.**
  2. **apply\_weighted\_average\_filter():**
     + **Applies user-defined weights to buffer values.**
     + **Returns the weighted sum.**
  3. **apply\_saturation():**
     + **Clamps values to a valid range (e.g., 0–500 cm).**
* **Key Variables:**
  1. **buffer[]: Stores historical values for filtering.**
  2. **weights[]: Predefined weights for averaging (e.g., [0.1, 0.2, 0.4, 0.2, 0.1]).**

1. **Printf\_redirect.cpp**

**A diagram of a computer program

AI-generated content may be incorrect.**

**Figure 3.4: Flowchart for** *printf\_redirect .cpp*

**Description**

* **Purpose: Redirects printf output to Arduino Serial.**
* **Functions:**
  1. **init\_printf\_redirect():**
     + **Calls fdevopen() to link printf to serial\_putchar.**
  2. **serial\_putchar():**
     + **Low-level function to write a character to Serial.**
  3. **serial\_printf() (fallback):**
     + **Formats strings using vsnprintf and sends via Serial.print().**

1. **ultrasonic.cpp**

A screenshot of a computer flowchart

AI-generated content may be incorrect.

*Figure 3.5: Flowchart for ultrasonic.cpp*

**Description**

* **Purpose: Measures distance using an ultrasonic sensor (HC-SR04).**
* **Functions:**
  1. **setup\_ultrasonic\_sensor():**
     + **Configures TRIGGER\_PIN (output) and ECHO\_PIN (input).**
  2. **measure\_distance():**
     + **Sends a 10µs trigger pulse.**
     + **Measures echo duration with pulseIn().**
     + **Calculates distance: distance = (pulse\_duration × SOUND\_SPEED) / 2.**
* **Key Constants:**
  1. **SOUND\_SPEED: 0.0343 cm/µs (speed of sound).**
  2. **MAX\_DISTANCE\_CM: Timeout threshold (e.g., 30ms → ~500 cm).**
* **Results**
* A circuit board with two wires

  AI-generated content may be incorrect.
* *Figure 3.6: Physical result*

*A screenshot of a computer

AI-generated content may be incorrect.*

*Figure 3.7: Console result*

**Conclusions**

After completing this laboratory work, the following results were achieved:

A functional **sensor data acquisition and display system** was successfully implemented, enabling real-time measurement, signal processing, and reporting using **FreeRTOS** and **STDIO**.

* The application correctly acquires **ultrasonic sensor data**, processes it into meaningful measurements (distance in cm), and displays the results on **Serial and/or LCD** at fixed intervals.
* A **modular architecture** was employed, separating functionalities into distinct tasks: **sensor acquisition (100ms), signal management (mutex-protected), and display (500ms)**.
* The system was tested on a microcontroller platform (e.g., Arduino with FreeRTOS) and validated for **real-time performance** and **data accuracy**.
* **Error handling** was integrated to detect invalid sensor readings (e.g., negative distances), triggering an **error state** for robustness.

**System Performance Analysis**

The system demonstrated **stable and efficient operation**, with reliable sensor sampling and reporting. Key performance aspects include:

* **Timing Precision:**
  + Sensor data is acquired every **100ms** using vTaskDelayUntil() for minimal jitter.
  + Display updates occur every **500ms** without blocking acquisition.
* **Data Integrity:**
  + Shared signals (e.g., distance, state) are protected by a **mutex**, preventing race conditions.
  + Invalid readings (e.g., out-of-range values) automatically trigger an **error state**.
* **Scalability:**
  + The modular design allows easy integration of additional sensors (e.g., temperature, humidity) or output devices (e.g., Wi-Fi modules).

**Identified Limitations**

Despite its functionality, the system has areas for improvement:

1. **Sensor Error Recovery** – The system detects errors but lacks an auto-recovery mechanism (e.g., retry logic).
2. **Limited Output Channels** – Data is only displayed on Serial; adding **LCD or wireless (Wi-Fi/MQTT)** would enhance usability.
3. **Fixed Sampling Rate** – The 100ms interval is hardcoded; a **user-configurable rate** (e.g., via buttons) would improve flexibility.
4. **Power Efficiency** – No low-power modes are implemented, which could be critical for battery-operated deployments.

**Impact of the Technology in Real-World Applications**

This project demonstrates principles applicable to:

* **Industrial Monitoring:**
  + Ultrasonic sensors measure tank levels or object proximity, with FreeRTOS ensuring real-time data logging.
* **Smart Agriculture:**
  + Soil moisture or light sensors could replace the ultrasonic module, with data transmitted to a cloud dashboard.
* **Home Automation:**
  + The system could be extended to control actuators (e.g., lights, alarms) based on sensor thresholds.
* **IoT Prototyping:**
  + Adding an ESP32 Wi-Fi module would enable remote monitoring via MQTT/HTTP.

**Improvement Suggestions**

To enhance the system, consider:

1. **Dynamic Sampling Rates** – Allow runtime adjustment (e.g., via Serial commands) for adaptive sensing.
2. **Wireless Integration** – Use **ESP-NOW or LoRa** for remote data transmission.
3. **Advanced Error Handling** – Implement sensor calibration or auto-retry on failure.
4. **Power Optimization** – Integrate FreeRTOS tickless idle mode for battery savings.
5. **Multi-Sensor Support** – Expand the signal manager to handle multiple sensor inputs (e.g., temperature + distance).

By addressing these points, the system could evolve into a **versatile embedded platform** for real-world IoT and automation applications.

**Final Remarks**

This lab successfully achieved its objectives, providing a **foundation for real-time sensor systems** with FreeRTOS. The modular design, mutex-protected data, and structured reporting make it a **reproducible template** for future embedded projects.

**Next Steps:**

* Integrate Wi-Fi (ESP32) for cloud connectivity.
* Develop a GUI dashboard (e.g., Node-RED) for visualization.
* Benchmark power consumption and optimize for low-energy scenarios.

### Note on AI Tool Usage

During the drafting of this report, the author used ChatGPT for generating and structuring the content. The resulting information was reviewed, validated, and adjusted according to the requirements of the laboratory work, ensuring technical accuracy and clarity of explanations.  
The use of this AI tool was aimed at structuring and optimizing the presentation of information without replacing personal analysis and understanding of the subject.

**Bibliography**

1. Official Arduino Documentation
   * Arduino Reference – Serial Communication  
     https://www.arduino.cc/reference/en/#communication
   * Arduino Mega 1280 Pinout & Datasheet  
     https://docs.arduino.cc/hardware/mega-1280
2. PlatformIO Official Documentation
   * PlatformIO for Arduino Development  
     https://docs.platformio.org/en/latest/platforms/atmelavr.html
3. TUM Courses
   * Introducere în Sistemele Embedded și Programarea Microcontrolerelor
   * Principiile comunicației seriale și utilizarea interfeței UART

**Appendix**

1. **GitHub**: <https://github.com/KaBoomKaBoom/ES_Labs.git>
2. Main.cpp

/\*\*

 \* Ultrasonic Sensor Application with FreeRTOS for Arduino Uno

 \* Main file: Contains setup, loop and main task

 \*/

 #include <Arduino.h>

 #include <Arduino\_FreeRTOS.h>

 #include <semphr.h>

 #include <stdio.h>

 #include "ultrasonic.h"

 #include "filter.h"

 #include "printf\_redirect.h"

 // Task parameters

 #define STACK\_SIZE          128     // Reduced stack size for Arduino Uno

 #define ACQUISITION\_PERIOD  100     // Acquisition period in milliseconds

 // Signal conditioning parameters

 #define SALT\_PEPPER\_WINDOW  5       // Window size for salt and pepper filter

 #define WEIGHTED\_AVG\_WINDOW 5       // Reduced window size for Arduino memory constraints

 // Filter buffers

 static float salt\_pepper\_buffer[SALT\_PEPPER\_WINDOW];

 static float weighted\_avg\_buffer[WEIGHTED\_AVG\_WINDOW];

 // Weights for weighted average (must sum to 1.0)

 static const float weights[WEIGHTED\_AVG\_WINDOW] = {0.1, 0.2, 0.4, 0.2, 0.1};

 // Function prototypes

 void ultrasonic\_distance\_task(void \*pvParameters);

 // Mutex for data access protection

 SemaphoreHandle\_t data\_mutex;

 void setup() {

   // Initialize serial communication

   Serial.begin(115200);

   while (!Serial) {

     ; // Wait for serial port to connect

   }

   // Initialize printf redirection

   init\_printf\_redirect();

   printf("Ultrasonic Sensor Application Starting...\n");

   // Create mutex

   data\_mutex = xSemaphoreCreateMutex();

   // Initialize hardware

   setup\_ultrasonic\_sensor();

   // Initialize filter buffers

   for (int i = 0; i < SALT\_PEPPER\_WINDOW; i++) {

     salt\_pepper\_buffer[i] = 0.0;

   }

   for (int i = 0; i < WEIGHTED\_AVG\_WINDOW; i++) {

     weighted\_avg\_buffer[i] = 0.0;

   }

   // Create task

   xTaskCreate(

     ultrasonic\_distance\_task,       // Task function

     "ultrasonic",                   // Task name

     STACK\_SIZE,                     // Stack size

     NULL,                           // Parameters

     1,                              // Priority

     NULL                            // Task handle

   );

   printf("Application initialized successfully\n");

   // Start the scheduler

   vTaskStartScheduler();

 }

 void loop() {

   // Empty. Things are done in Tasks.

 }

 /\*\*

  \* Task for ultrasonic distance measurement and processing

  \*/

 void ultrasonic\_distance\_task(void \*pvParameters) {

   (void) pvParameters;

   TickType\_t last\_wake\_time;

   const TickType\_t period = pdMS\_TO\_TICKS(ACQUISITION\_PERIOD);

   // Initialize the last\_wake\_time variable with the current time

   last\_wake\_time = xTaskGetTickCount();

   float raw\_distance, filtered\_distance1, filtered\_distance2, final\_distance;

   while (1) {

     // Get raw distance measurement

     raw\_distance = measure\_distance();

     // Apply salt and pepper filter

     filtered\_distance1 = apply\_salt\_pepper\_filter(raw\_distance, salt\_pepper\_buffer, SALT\_PEPPER\_WINDOW);

     // Apply weighted average filter

     filtered\_distance2 = apply\_weighted\_average\_filter(filtered\_distance1, weighted\_avg\_buffer, weights, WEIGHTED\_AVG\_WINDOW);

     // Apply saturation to keep values in valid range

     final\_distance = apply\_saturation(filtered\_distance2, MIN\_DISTANCE\_CM, MAX\_DISTANCE\_CM);

     // Acquire mutex to print data

     if (xSemaphoreTake(data\_mutex, portMAX\_DELAY) == pdTRUE) {

       // Print results using printf

       printf("Distance Measurements:\n");

       printf("  Raw: %.2f cm\n", raw\_distance);

       printf("  After Salt & Pepper: %.2f cm\n", filtered\_distance1);

       printf("  After Weighted Avg: %.2f cm\n", filtered\_distance2);

       printf("  Final (Saturated): %.2f cm\n\n", final\_distance);

       // Release mutex

       xSemaphoreGive(data\_mutex);

     }

     // Wait precisely for the next cycle using vTaskDelayUntil

     vTaskDelayUntil(&last\_wake\_time, period);

   }

 }

1. filter.cpp

/\*\*

 \* Digital filters module implementation file

 \*/

 #include <Arduino.h>

 #include <string.h>

 #include "filter.h"

 /\*\*

  \* Salt and Pepper filter (median filter)

  \* Removes impulsive noise by taking the median value from a window

  \*/

 float apply\_salt\_pepper\_filter(float new\_value, float \*buffer, int window\_size) {

   float sorted\_values[window\_size];

   // Shift values in buffer

   for (int i = 0; i < window\_size - 1; i++) {

     buffer[i] = buffer[i + 1];

   }

   buffer[window\_size - 1] = new\_value;

   // Copy values to sorting array

   memcpy(sorted\_values, buffer, sizeof(float) \* window\_size);

   // Perform simple bubble sort (sufficient for small arrays)

   for (int i = 0; i < window\_size - 1; i++) {

     for (int j = 0; j < window\_size - i - 1; j++) {

       if (sorted\_values[j] > sorted\_values[j + 1]) {

         float temp = sorted\_values[j];

         sorted\_values[j] = sorted\_values[j + 1];

         sorted\_values[j + 1] = temp;

       }

     }

   }

   // Return median value

   return sorted\_values[window\_size / 2];

 }

 /\*\*

  \* Weighted Average filter

  \* Applies weighted average to smooth the signal

  \*/

 float apply\_weighted\_average\_filter(float new\_value, float \*buffer, const float \*weights, int window\_size) {

   // Shift values in buffer

   for (int i = 0; i < window\_size - 1; i++) {

     buffer[i] = buffer[i + 1];

   }

   buffer[window\_size - 1] = new\_value;

   // Apply weighted average

   float weighted\_sum = 0.0;

   for (int i = 0; i < window\_size; i++) {

     weighted\_sum += buffer[i] \* weights[i];

   }

   return weighted\_sum;

 }

 /\*\*

  \* Apply saturation to ensure value is within valid range

  \*/

 float apply\_saturation(float value, float min\_val, float max\_val) {

   if (value < min\_val) {

     return min\_val;

   } else if (value > max\_val) {

     return max\_val;

   } else {

     return value;

   }

 }

1. ultrasonic.cpp

/\*\*

 \* Ultrasonic sensor module implementation file

 \*/

 #include <Arduino.h>

 #include "ultrasonic.h"

 /\*\*

  \* Setup function for ultrasonic sensor pins

  \*/

 void setup\_ultrasonic\_sensor(void) {

   // Configure trigger pin as output

   pinMode(TRIGGER\_PIN, OUTPUT);

   digitalWrite(TRIGGER\_PIN, LOW);

   // Configure echo pin as input

   pinMode(ECHO\_PIN, INPUT);

 }

 /\*\*

  \* Measure distance using ultrasonic sensor

  \* @return Distance in centimeters

  \*/

 float measure\_distance(void) {

   // Variables to store pulse timing

   unsigned long pulse\_start, pulse\_end;

   float pulse\_duration, distance;

   // Clear trigger pin

   digitalWrite(TRIGGER\_PIN, LOW);

   delayMicroseconds(2);

   // Send a 10µs pulse to trigger

   digitalWrite(TRIGGER\_PIN, HIGH);

   delayMicroseconds(10);

   digitalWrite(TRIGGER\_PIN, LOW);

   // Measure the length of echo pulse

   pulse\_duration = pulseIn(ECHO\_PIN, HIGH, 30000); // Timeout after 30ms

   // If timeout occurred, return max distance

   if (pulse\_duration == 0) {

     return MAX\_DISTANCE\_CM;

   }

   // Calculate distance: pulse duration \* speed of sound / 2

   // (divided by 2 because sound travels to object and back)

   distance = (pulse\_duration \* SOUND\_SPEED) / 2.0;

   return distance;

 }

1. printf\_redirect.cpp

/\*\*

 \* Printf redirection module implementation file

 \*/

 #include <Arduino.h>

 #include <stdio.h>

 #include <stdarg.h>

 #include "printf\_redirect.h"

 /\*\*

  \* Initialize printf redirection to Serial

  \*/

 void init\_printf\_redirect(void) {

   // Redirect stdout to Serial

   fdevopen(&serial\_putchar, NULL);

 }

 /\*\*

  \* Implementation of putchar for printf

  \*/

 int serial\_putchar(char c, FILE \*stream) {

   Serial.write(c);

   return 0;

 }

 /\*\*

  \* Custom implementation for printf that redirects to Serial

  \* This is a backup method if fdevopen doesn't work

  \*/

 int serial\_printf(const char \*format, ...) {

   char buf[128]; // Buffer for formatted string

   va\_list args;

   va\_start(args, format);

   vsnprintf(buf, sizeof(buf), format, args);

   va\_end(args);

   Serial.print(buf);

   return strlen(buf);

 }