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. 3.1

**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 fundamental in modern applications, where real-time data acquisition and processing are critical. Microcontrollers (MCUs) equipped with analog and digital sensors enable precise environmental monitoring, automation, and control systems.

This laboratory work focuses on signal acquisition from sensors (analog or digital), processing the data, and displaying it on an LCD or Serial terminal using FreeRTOS for task scheduling and STDIO for reporting. The system demonstrates modularity, real-time monitoring, and efficient task management—essential in industrial automation, IoT, and smart sensing applications.

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

Microcontroller (ESP32/STM32/Arduino with FreeRTOS) – Central unit for sensor data acquisition, processing, and task scheduling.

* Sensor (Analog/Digital) – Selected from 37Sensors (e.g., temperature sensor, motion sensor, etc.)
* Breadboard & Jumper Wires – For prototyping and interconnections.
* Power Supply (USB/Battery) – Provides stable voltage to the MCU and peripherals.

**Software Components**

* Development Environment (PlatformIO/Arduino IDE/STM32CubeIDE) – For coding, compiling, and flashing the firmware.
* FreeRTOS – Manages task scheduling (vTaskDelay, vTaskDelayUntil) for periodic sensor readings and display updates.
* STDIO Library (printf, scanf) – Facilitates Serial communication and LCD output for reporting sensor data.
* Sensor-Specific Libraries (e.g., DHT.h for temperature, Wire.h for I2C sensors) – Enables seamless sensor interfacing.

1. **System Architecture Explanation and Solution Justification**

The system follows a modular FreeRTOS-based architecture, where tasks are scheduled for:

**Sensor Data Acquisition** – Reads analog/digital signals at fixed intervals.

**Signal Processing** – Scales raw data into physical values (e.g., °C, lux).

**Data Display** – Outputs processed data to LCD/Serial every 500ms.

A screenshot of a computer program

AI-generated content may be incorrect.

**Why This Solution?**

* Real-Time Performance – FreeRTOS ensures precise timing for sensor sampling.
* Modularity – Separates acquisition, processing, and display for scalability.
* Efficient Resource Use – Non-blocking delays prevent CPU overload.
* Flexibility – Supports multiple sensor types (analog/digital).

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 wires connected to it

AI-generated content may be incorrect.**

*Figure 2.2 Component scheme*

A computer circuit board with wires

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 diagram of software development

AI-generated content may be incorrect.

*Figure 2.4 Flowchart*

**Explanation:**

* **Initialization:** Hardware (sensor, Serial) and FreeRTOS tasks are set up.
* **Acquisition Task:**
  + Reads sensor data (sensor\_read\_distance).
  + Updates shared signals (signal\_manager\_update).
  + Uses vTaskDelayUntil for precise 100ms intervals.
* **Display Task:**
  + Retrieves signals (signal\_manager\_get\_signals).
  + Prints formatted report (printReport) every 500ms.

A diagram of a system

AI-generated content may be incorrect.

*Figure 2.4 FSM diagram*

**States:**

* **SYSTEM\_IDLE:** Default state (no active measurements).
* **SYSTEM\_MEASURING:** Valid sensor data received (distance ≥ 0).
* **SYSTEM\_ERROR:** Invalid data (distance < 0 or sensor failure).

**Transitions:**

* **Valid Data:** Moves from IDLE to MEASURING.
* **Error Condition:** Invalid distance triggers ERROR state.
* **Recovery:** Manual reset or valid data resumes MEASURING/IDLE.

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 program

AI-generated content may be incorrect.**

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

**Description**

* **Initialization:**
  + Sets up Serial communication (Serial.begin(115200).
  + Redirects printf to Serial via fdev\_setup\_stream.
* **Hardware Setup:**
  + Calls sensor\_init() (configures HC-SR04 pins).
  + Initializes signal manager (signal\_manager\_init() creates mutex).
* **RTOS Tasks:**
  + Creates two tasks:
    - acquisitionTask (priority 2, 128B stack).
    - displayTask (priority 1, 256B stack).
  + Starts FreeRTOS scheduler (vTaskStartScheduler).
* **Loop:** Empty (scheduler handles execution).

1. **sensor.cpp**

A diagram of a device

AI-generated content may be incorrect.

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

**Description**

* **sensor\_init():**
  + No explicit setup (handled by NewPing library).
* **sensor\_read\_distance():**
  1. Triggers ultrasonic pulse (sonar.ping\_cm()).
  2. Measures echo time → converts to cm.
  3. Validates reading:
     + Returns MAX\_DISTANCE (400cm) if no echo.
     + Else returns actual distance (0–400cm).

1. **signal\_manager.cpp**

**A screenshot of a computer flowchart

AI-generated content may be incorrect.**

**Figure 3.4: Flowchart for** *signal\_manager.cpp*

**Description**

* **signal\_manager\_init():**
  + Creates mutex (xSemaphoreCreateMutex) to protect shared SystemSignals.
* **signal\_manager\_update():**
  1. Locks mutex → updates distance, count, timestamp.
  2. Calls updateSystemState() to set state and errorCode.
  3. Releases mutex.
* **signal\_manager\_get\_signals():**
  + Safely copies SystemSignals using mutex.
* **updateSystemState():**
  + Sets state based on distance:
    - ERROR if distance < 0.
    - MEASURING otherwise (with error code 0).

1. **display.cpp**

A diagram of a software system

AI-generated content may be incorrect.

*Figure 3.5: Flowchart for display.cpp*

**Description**

* **printReport():**
  + Formats and prints SystemSignals via printf:
    - State (IDLE/MEASURING/ERROR).
    - Distance, measurement count, error code (if any).
* **displayTask():**
  1. Starts with 50ms offset (to avoid collision with acquisition task).
  2. Every 500ms:
     + Retrieves signals (signal\_manager\_get\_signals).
     + Calls printReport().

**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 Data Acquisition and Display using FreeRTOS

 \*

 \* Main file - Program initialization and entry point

 \*/

 #include <Arduino\_FreeRTOS.h>

 #include <stdio.h>

 #include "sensor.h"

 #include "signal\_manager.h"

 #include "display.h"

 // Task handles

 TaskHandle\_t acquisitionTaskHandle;

 TaskHandle\_t displayTaskHandle;

 // Required for printf redirection to Serial

 int serialPutchar(char c, FILE \*) {

   Serial.write(c);

   return c;

 }

 // Setup printf

 FILE serial\_stdout;

 void setup() {

   // Initialize serial communication

   Serial.begin(115200);

   // Set up printf to output to Serial

   fdev\_setup\_stream(&serial\_stdout, serialPutchar, NULL, \_FDEV\_SETUP\_WRITE);

   stdout = &serial\_stdout;

   // Print startup message

   printf("\nUltrasonic Sensor System Starting...\n");

   // Initialize sensor

   sensor\_init();

   // Initialize signal manager

   signal\_manager\_init();

   // Create FreeRTOS tasks

   xTaskCreate(

     acquisitionTask,          // Task function

     "AcquisitionTask",        // Task name

     128,                      // Stack size

     NULL,                     // Parameters

     2,                        // Priority (higher number = higher priority)

     &acquisitionTaskHandle    // Task handle

   );

   xTaskCreate(

     displayTask,              // Task function

     "DisplayTask",            // Task name

     256,                      // Stack size (larger for display task due to printf)

     NULL,                     // Parameters

     1,                        // Priority (lower than acquisition task)

     &displayTaskHandle        // Task handle

   );

   // Start the scheduler

   vTaskStartScheduler();

 }

 void loop() {

   // Empty, as FreeRTOS takes control

 }

1. Signal\_manager.cpp

/\*

 \* signal\_manager.cpp

 \*

 \* Implementation file for signal and state management

 \*/

 #include "signal\_manager.h"

 #include "sensor.h"

 #include <string.h>

 // Global system signals

 static SystemSignals signals = {0, SYSTEM\_IDLE, 0, 0, 0};

 // Mutex for protecting access to shared signals

 static SemaphoreHandle\_t signalsMutex;

 /\*\*

  \* Update system state based on sensor data

  \*/

 static void updateSystemState(SystemSignals \*signals) {

   // Check for sensor errors

   if (signals->distance < 0) {

     signals->state = SYSTEM\_ERROR;

     signals->errorCode = 1; // Error code 1: Invalid distance reading

   }

   // Check if object is too close (less than 10cm)

   else if (signals->distance < 10) {

     signals->state = SYSTEM\_MEASURING;

     signals->errorCode = 0; // No error

   }

   // Normal operation

   else {

     signals->state = SYSTEM\_MEASURING;

     signals->errorCode = 0; // No error

   }

 }

 void signal\_manager\_init() {

   // Create mutex for signals

   signalsMutex = xSemaphoreCreateMutex();

   // Initialize signals

   signals.distance = 0;

   signals.state = SYSTEM\_IDLE;

   signals.errorCode = 0;

   signals.measurementCount = 0;

   signals.lastMeasurementTime = 0;

 }

 void signal\_manager\_update(float distance) {

   // Acquire mutex before updating shared signals

   if (xSemaphoreTake(signalsMutex, pdMS\_TO\_TICKS(10)) == pdTRUE) {

     signals.distance = distance;

     signals.measurementCount++;

     signals.lastMeasurementTime = xTaskGetTickCount();

     // Update system state based on sensor data

     updateSystemState(&signals);

     // Release mutex

     xSemaphoreGive(signalsMutex);

   }

 }

 bool signal\_manager\_get\_signals(SystemSignals \*dest) {

   bool success = false;

   // Acquire mutex before reading shared signals

   if (xSemaphoreTake(signalsMutex, pdMS\_TO\_TICKS(10)) == pdTRUE) {

     // Copy signals to destination

     memcpy(dest, &signals, sizeof(SystemSignals));

     // Release mutex

     xSemaphoreGive(signalsMutex);

     success = true;

   }

   return success;

 }

 void acquisitionTask(void \*pvParameters) {

   TickType\_t xLastWakeTime;

   const TickType\_t xFrequency = pdMS\_TO\_TICKS(100); // 100ms sampling rate

   // Initialize the xLastWakeTime variable with the current time

   xLastWakeTime = xTaskGetTickCount();

   for (;;) {

     // Wait for the next cycle with precise timing

     vTaskDelayUntil(&xLastWakeTime, xFrequency);

     // Read distance from sensor

     float distance = sensor\_read\_distance();

     // Update signals with new reading

     signal\_manager\_update(distance);

   }

 }

1. Sensor.cpp

/\*

 \* sensor.cpp

 \*

 \* Implementation file for ultrasonic sensor functionality

 \*/

 #include "sensor.h"

 #include <NewPing.h>

 // Global objects

 static NewPing sonar(TRIGGER\_PIN, ECHO\_PIN, MAX\_DISTANCE);

 void sensor\_init() {

   // HC-SR04 requires no special initialization

   // Pins are configured by the NewPing library

 }

 float sensor\_read\_distance() {

   // Measure distance

   float distance = sonar.ping\_cm();

   // If distance is 0, sensor might be out of range or there's an error

   if (distance == 0) {

     distance = MAX\_DISTANCE; // Set to max distance if no echo received

   }

   return distance;

 }

1. Display.cpp

/\*

 \* display.cpp

 \*

 \* Implementation file for display functionality

 \*/

 #include "display.h"

 #include <stdio.h>

 void printReport(const SystemSignals \*signals) {

   // Print header

   printf("\n========== SYSTEM REPORT ==========\n");

   // Print current time

   printf("Time: %lu ms\n", (unsigned long)xTaskGetTickCount());

   // Print system state

   printf("State: ");

   switch (signals->state) {

     case SYSTEM\_IDLE:

       printf("IDLE\n");

       break;

     case SYSTEM\_MEASURING:

       printf("MEASURING\n");

       break;

     case SYSTEM\_ERROR:

       printf("ERROR\n");

       break;

     default:

       printf("UNKNOWN\n");

   }

   // Print distance

   printf("Distance: %.2f cm\n", signals->distance);

   // Print measurement count

   printf("Measurements: %lu\n", (unsigned long)signals->measurementCount);

   // Print error code if any

   if (signals->errorCode > 0) {

     printf("Error Code: %d\n", signals->errorCode);

   }

   printf("====================================\n");

 }

 void displayTask(void \*pvParameters) {

   // Add a small delay to offset from acquisition task

   vTaskDelay(pdMS\_TO\_TICKS(50));

   const TickType\_t xFrequency = pdMS\_TO\_TICKS(500); // 500ms update rate

   SystemSignals localSignals;

   for (;;) {

     // Get a copy of the current signals

     if (signal\_manager\_get\_signals(&localSignals)) {

       // Print report using printf

       printReport(&localSignals);

     }

     // Wait for the next cycle

     vTaskDelay(xFrequency);

   }

 }

1. Signal\_manager.h

/\*

 \* signal\_manager.h

 \*

 \* Header file for signal and state management

 \*/

 #ifndef SIGNAL\_MANAGER\_H

 #define SIGNAL\_MANAGER\_H

 #include <Arduino.h>

 #include <Arduino\_FreeRTOS.h>

 #include <semphr.h>

 // System state definitions

 typedef enum {

   SYSTEM\_IDLE,

   SYSTEM\_MEASURING,

   SYSTEM\_ERROR

 } SystemState;

 // Signal structure to hold sensor data and system state

 typedef struct {

   float distance;         // Distance measured by ultrasonic sensor in cm

   SystemState state;      // Current system state

   uint8\_t errorCode;      // Error code (0 = no error)

   uint32\_t measurementCount; // Count of measurements taken

   TickType\_t lastMeasurementTime; // Time of last measurement

 } SystemSignals;

 /\*\*

  \* Initialize the signal manager

  \*/

 void signal\_manager\_init();

 /\*\*

  \* Update system signals with new sensor data

  \*

  \* @param distance the newly measured distance

  \*/

 void signal\_manager\_update(float distance);

 /\*\*

  \* Get a copy of the current system signals

  \*

  \* @param dest pointer to a SystemSignals struct to be filled with current values

  \* @return true if successful, false if mutex could not be acquired

  \*/

 bool signal\_manager\_get\_signals(SystemSignals \*dest);

 /\*\*

  \* Task for acquiring data from the ultrasonic sensor

  \* Uses vTaskDelayUntil for precise timing

  \*/

 void acquisitionTask(void \*pvParameters);

 #endif // SIGNAL\_MANAGER\_H

1. Sensor.h

/\*

 \* sensor.h

 \*

 \* Header file for ultrasonic sensor functionality

 \*/

 #ifndef SENSOR\_H

 #define SENSOR\_H

 #include <Arduino.h>

 // Pin definitions for HC-SR04 ultrasonic sensor

 #define TRIGGER\_PIN 9

 #define ECHO\_PIN 10

 #define MAX\_DISTANCE 400 // Maximum distance to measure (in cm)

 /\*\*

  \* Initialize the ultrasonic sensor

  \*/

 void sensor\_init();

 /\*\*

  \* Read distance from ultrasonic sensor

  \*

  \* @return measured distance in cm, or MAX\_DISTANCE if no echo received

  \*/

 float sensor\_read\_distance();

 #endif // SENSOR\_H

1. Display.h

/\*

 \* display.h

 \*

 \* Header file for display functionality

 \*/

 #ifndef DISPLAY\_H

 #define DISPLAY\_H

 #include "signal\_manager.h"

 /\*\*

  \* Print formatted report using printf

  \*

  \* @param signals pointer to the system signals to be displayed

  \*/

 void printReport(const SystemSignals \*signals);

 /\*\*

  \* Task for displaying data on Serial

  \* Uses vTaskDelay for timing

  \*/

 void displayTask(void \*pvParameters);

 #endif // DISPLAY\_H